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

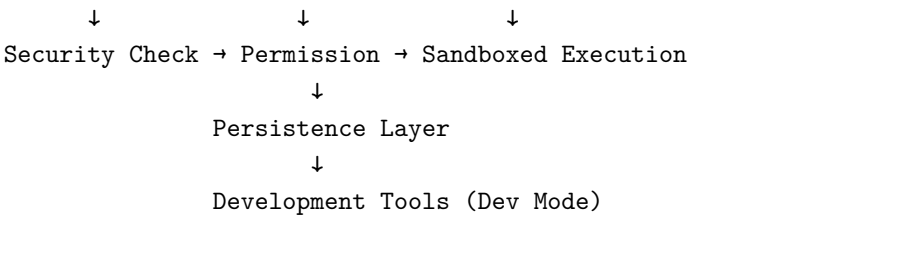
HMR (Vite/WP)	**Source Maps** (Debugging)	**Error Overlay** (Dev Mode)
State Inspector (DevTools)	**Performance Profiling**	**Network Monitoring** (Extension)

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);
  }
}
```

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
MobX	Observable	Very reactive; Automatic tracking; Powerful	Large bundle; Magic behavior; Learning curve	~16KB	Complex reactive apps

Library	Pattern	Pros	Cons	Bundle Size	Use Case
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
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

Platform	State Pattern	Implementation
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
  )
);
```

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 in the BI Dashboard Framework. The Plugin Loader serves as the extensibility engine of the application, enabling the framework to evolve beyond its core capabilities by dynamically loading, managing, and executing third-party extensions at runtime. It orchestrates the complete plugin lifecycle—from discovery and loading to activation, hot-swapping, and graceful shutdown—while maintaining system stability and security.

This component enables developers to extend dashboard functionality with custom visualizations, data connectors, transformation pipelines, UI components, and business logic without modifying the core codebase. The loader supports hot module replacement for seamless development workflows, manages complex plugin dependency graphs to ensure correct load order, provides sandboxed execution contexts to isolate plugin code from the core system, and enforces permission-based access control to protect sensitive APIs.

By treating plugins as first-class citizens with well-defined lifecycle hooks (init, activate, deactivate, destroy), the loader ensures predictable behavior, proper resource cleanup, and the ability to enable/disable extensions on-the-fly without requiring application restarts. This architecture transforms the BI Dashboard from a static application into a living platform that can be customized and extended to meet diverse business requirements while maintaining performance, security, and developer experience.

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

```typescript
// 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
    }
  }
}
```

Chapter 3

Extension System

The Extension System represents Layer 3 of the framework architecture, providing the mechanisms and interfaces through which developers extend, customize, and enhance the BI Dashboard beyond its core capabilities. This layer embodies the framework's "Maximum Extensibility" philosophy by offering multiple pathways for extension: a custom Domain-Specific Language (DSL) for declarative, safe UI composition; full JavaScript/TypeScript support for programmatic control; and a rich set of extension points covering UI components, commands, keybindings, data transformations, themes, and layouts.

The system is designed to accommodate developers of all skill levels—from business analysts using the DSL to create simple dashboards, to advanced developers building complex visualizations with React and external libraries. Each extension type is sandboxed for security, versioned for compatibility, and hot-reloadable for rapid development.

Extensions can register new chart types, contribute commands to the palette, define custom keybindings, hook into application lifecycle events, override default behaviors through the advice system, and even extend other extensions through a composable plugin architecture. This multi-layered approach ensures that the framework remains accessible to non-technical users while providing the depth and flexibility that power users demand, all while maintaining security boundaries, performance characteristics, and a consistent developer experience across different extension methods.

3.1 Extension Languages

3.1.1 Custom DSL

3.1.1.1 Purpose

Declarative, safe UI composition for the BI Dashboard Framework. The Custom DSL provides a high-level, domain-specific language tailored specifically for creating dashboard layouts, visualizations, and data-driven interfaces without requiring deep programming knowledge. This language serves as the primary extension method for business analysts, data scientists, and non-technical users who need to build sophisticated dashboards quickly and safely.

The DSL abstracts away the complexity of React components, state management, and event handling, presenting instead a clean, readable syntax that focuses on what to display rather than how to implement it. By design, the DSL is intentionally constrained to safe operations—preventing arbitrary code execution, file system access, or network requests—making it ideal for multi-tenant environments and scenarios where untrusted users need to create custom dashboards.

The language compiles directly to optimized React components, ensuring that DSL-defined dashboards perform identically to hand-coded JavaScript equivalents while maintaining type safety through static analysis. This approach democratizes dashboard creation, enabling rapid prototyping and iteration through hot-reloading, while the compiler catches errors at build time rather than runtime.

The DSL supports common BI patterns out-of-the-box—grid layouts, chart configurations, data queries, filters, and interactions—reducing boilerplate and allowing users to express complex dashboard logic in just a few lines of declarative code.

3.1.1.2 Features

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

3.1.1.3 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)
  }
}
```

3.2 JavaScript Extensions

3.2.1 Purpose

Full programmatic control for advanced use cases in the BI Dashboard Framework. JavaScript Extensions provide unrestricted access to the framework's extension API, enabling advanced developers to build sophisticated features that go beyond the capabilities of the declarative DSL. This extension method is designed for scenarios requiring complex business logic, custom algorithms, integration with external services, advanced data transformations, or novel visualization techniques that cannot be expressed through the DSL's constrained syntax.

JavaScript extensions have full access to the React ecosystem, allowing developers to leverage popular libraries like D3.js for custom visualizations, date-fns for temporal calculations, or lodash for data manipulation. Unlike the DSL, JavaScript extensions can implement stateful components with complex lifecycle management, perform asynchronous operations like API calls and database queries, subscribe to real-time data streams, and interact with browser APIs for features like clipboard access or file downloads.

While this power comes with increased responsibility—JavaScript extensions run in a sandboxed environment with permission-based access control to prevent malicious code from compromising the application—developers retain the flexibility to build virtually any feature imaginable. The framework provides a comprehensive extension API that exposes core services (component registry, event bus, state store, command palette) while maintaining security boundaries, ensuring that even powerful JavaScript extensions cannot break the application or access unauthorized resources.

This dual-language approach—safe DSL for common cases, powerful JavaScript for advanced scenarios—strikes the optimal balance between accessibility and capability.

3.2.2 Features

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

- Sandboxed execution

3.2.3 Extension Points

The framework provides multiple extension points for customizing every aspect of the system, enabling developers to inject custom behavior, UI elements, and functionality at strategic integration points throughout the application lifecycle. These extension points represent the framework's commitment to the "Maximum Extensibility" philosophy, offering well-defined interfaces where plugins can hook into core systems without modifying the underlying codebase.

Extension points span the entire application stack: **UI Components** for custom visualizations and widgets, **Commands** for registering actions in the command palette, **Keybindings** for keyboard shortcuts, **Hooks** for intercepting and augmenting function calls, **Advice** for wrapping existing methods with additional behavior, **Themes** for visual customization, **Layouts** for dashboard arrangement patterns, **Data Transformations** for processing query results, **Event Listeners** for reacting to application state changes, and **Lifecycle Hooks** for plugin initialization and cleanup.

Each extension point is designed with composability in mind—multiple extensions can contribute to the same point without conflicts, thanks to priority systems, namespacing, and conflict resolution strategies. The framework guarantees that extensions registered at these points receive stable, versioned APIs that won't break between releases, and provides comprehensive TypeScript definitions for type-safe extension development.

This architecture allows the core framework to remain minimal while supporting unlimited extensibility, as new capabilities can be added through extensions rather than core modifications, ensuring the system scales gracefully as requirements evolve.

3.2.4 UI Components

UI Components represent the most visible and frequently used extension point in the BI Dashboard Framework, enabling developers to create custom visualizations, interactive widgets, data displays, and user interface elements that seamlessly integrate with the dashboard ecosystem. This extension point is fundamental to the framework's value proposition—while the core provides essential charts (line, bar, pie, scatter) and standard widgets (tables, cards, metrics), the real power emerges when organizations can build domain-specific visualizations tailored to their unique business needs.

Component extensions can range from simple data displays (KPI cards, progress bars, status indicators) to sophisticated interactive visualizations (network graphs, geospatial maps, custom D3.js charts, WebGL-accelerated renderings) to complex composite widgets (filterable data tables, drill-down hierarchies, multi-step forms).

The framework supports multiple component authoring approaches—React components for developers familiar with the ecosystem, Web Components for framework-agnostic reusability, declarative configurations for simple cases, and render functions for lightweight elements—ensuring that component creation is accessible regardless of technical background or architectural preferences.

All component extensions benefit from automatic integration with the framework's state management, event system, theme engine, and layout manager, meaning developers can focus on visualization logic rather than infrastructure concerns. Components receive reactive data updates, respond to theme changes, participate in the command palette, and can be drag-dropped onto dashboards through the visual builder, providing a first-class user experience identical to built-in components.

3.2.5 Extension Capabilities

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

3.2.6 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

3.2.7 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

3.2.8 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.

3.3 Commands & Command Palette

3.3.1 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

3.3.2 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

3.3.3 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

3.3.4 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

3.3.5 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

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

3.3.6 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

Algorithm	Approach	Pros	Cons	Best For
N-gram Based	Token matching	Language-aware; Good for text; Handles word order	Slower; More memory; Complex setup	Full-text search, documents

3.3.7 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
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

3.3.8 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)
  }
};
```

3.3.9 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

3.3.10 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`

3.3.11 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

3.3.12 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

3.3.13 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.

3.3.14 Extension Capabilities

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

3.3.15 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

3.3.16 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
VS Code	Commands API	• commands.registerCommand; • Command palette; • Keybinding integration

3.3.17 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.

3.4 Keybindings

3.4.1 Core Concepts

- **Global Keybindings**: System-wide keyboard shortcuts that remain active regardless of which component has focus or what mode the application is in. These bindings handle universal actions like opening the command palette (`Cmd/Ctrl+K`), creating new dashboards (`Cmd/Ctrl+N`), saving work (`Cmd/Ctrl+S`), or toggling the sidebar. Global keybindings have the lowest priority in the resolution hierarchy, ensuring they can be overridden by more specific context-aware bindings when necessary. They provide a consistent baseline of keyboard navigation that users can rely on throughout the application.
- **Mode-Specific Keybindings**: Context-aware keyboard shortcuts that activate only when specific conditions are met, such as when a particular component has focus (text editor, chart canvas, data table) or when the application is in a specific mode (edit mode, presentation mode, debug mode). For example, `Enter` might execute a query in the SQL editor but insert a new row in a data table, while `Escape` might exit edit mode in one context but close a modal in another. Mode-specific bindings have higher priority than global bindings, allowing the same keys to perform different actions based on context without conflicts.
- **Keymaps**: Hierarchical collections of keybinding definitions organized by scope, priority, and context, enabling systematic management of keyboard shortcuts across the application. Keymaps support inheritance (child keymaps can extend parent keymaps), composition (multiple keymaps can be active simultaneously), and priority-based conflict resolution (Local → Mode → Global). Extensions can contribute their own keymaps that integrate seamlessly with core bindings, and users can create custom keymaps to match their preferred workflows (Vim-style, Emacs-style, IDE-style). The keymap system ensures predictable behavior even when dozens of extensions register competing shortcuts.
- **Chord Support**: Multi-key sequences that expand the available keyboard shortcut space by allowing commands to be triggered by pressing keys in sequence rather than simultaneously. For example, `Ctrl+K Ctrl+S` (press Ctrl+K, release, then press Ctrl+K+S) or `g g` (press 'g' twice) enable hundreds of additional shortcuts without requiring obscure modifier combinations. Chord sequences include configurable timeouts (typically 1-2 seconds) to distinguish intentional sequences from accidental key presses, visual feedback showing the current chord state, and the ability to cancel incomplete sequences with `Escape`. This feature is essential for power users who need quick access to many commands without memorizing complex modifier combinations.
- **Customizable**: Users can rebind any key combination to any command through a visual keybinding editor, JSON configuration files, or the settings panel, enabling personalization for different workflows, accessibility needs, and muscle memory from other tools. The customization system validates bindings to prevent conflicts, warns about shadowed shortcuts, supports importing/exporting keybinding profiles, and provides search/filter capabilities to find and modify specific bindings. Custom keybindings persist across sessions via IndexedDB and can be synced across devices through cloud storage, ensuring a consistent experience. The system also supports resetting individual bindings or entire keymaps to defaults when needed.

3.4.2 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)

3.4.3 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

3.4.4 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

3.4.5 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.

3.4.6 Extension Capabilities

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

3.4.7 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

3.4.8 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

3.4.9 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' }
  ]
});
```

Note: Detailed keybinding architecture consolidated here from multiple sections.

3.5 Themes

3.5.1 Purpose

Themes represent the third most visible and frequently used extension point in the BI Dashboard Framework, enabling developers to create custom visualizations, interactive widgets, data displays, and user interface elements that seamlessly integrate with the dashboard ecosystem. This extension point is fundamental to the framework's value proposition—while

the core provides essential charts (line, bar, pie, scatter) and standard widgets (tables, cards, metrics), the real power emerges when organizations can build domain-specific visualizations tailored to their unique business needs.

Component extensions can range from simple data displays (KPI cards, progress bars, status indicators) to sophisticated interactive visualizations (network graphs, geospatial maps, custom D3.js charts, WebGL-accelerated renderings) to complex composite widgets (filterable data tables, drill-down hierarchies, multi-step forms).

The framework supports multiple component authoring approaches—React components for developers familiar with the ecosystem, Web Components for framework-agnostic reusability, declarative configurations for simple cases, and render functions for lightweight elements—ensuring that component creation is accessible regardless of technical background or architectural preferences.

All component extensions benefit from automatic integration with the framework's state management, event system, theme engine, and layout manager, meaning developers can focus on visualization logic rather than infrastructure concerns. Components receive reactive data updates, respond to theme changes, participate in the command palette, and can be drag-dropped onto dashboards through the visual builder, providing a first-class user experience identical to built-in components.

3.5.2 Extension Capabilities

- Color schemes and palettes
- Typography systems
- Component styling
- Dark/light mode variants
- Custom CSS variables

3.5.3 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

3.5.4 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

Platform	Theme System	Extension Method
Count.co	Theme settings	• Color customization; • Canvas themes; • CSS variables
tldraw	Theme system	• CSS variables; • Custom themes; • Dark/light mode; • Color overrides

3.5.5 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)'
  }
});
```

3.6 Layouts

3.6.1 Purpose

Layouts represent the fourth most visible and frequently used extension point in the BI Dashboard Framework, enabling developers to create custom visualizations, interactive widgets, data displays, and user interface elements that seamlessly integrate with the dashboard ecosystem. This extension point is fundamental to the framework's value proposition—while the core provides essential charts (line, bar, pie, scatter) and standard widgets (tables, cards, metrics), the real power emerges when organizations can build domain-specific visualizations tailored to their unique business needs.

Component extensions can range from simple data displays (KPI cards, progress bars, status indicators) to sophisticated interactive visualizations (network graphs, geospatial maps, custom D3.js charts, WebGL-accelerated renderings) to complex composite widgets (filterable data tables, drill-down hierarchies, multi-step forms).

The framework supports multiple component authoring approaches—React components for developers familiar with the ecosystem, Web Components for framework-agnostic reusability, declarative configurations for simple cases, and render functions for lightweight elements—ensuring that component creation is accessible regardless of technical background or architectural preferences.

All component extensions benefit from automatic integration with the framework's state management, event system, theme engine, and layout manager, meaning developers can focus on visualization logic rather than infrastructure concerns. Components receive reactive data updates, respond to theme changes, participate in the command palette, and can be drag-dropped onto dashboards through the visual builder, providing a first-class user experience identical to built-in components.

3.6.2 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

3.6.3 Extension Capabilities

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

3.6.4 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

3.6.5 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
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

3.6.6 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.

3.7 Hooks & Advice

3.7.1 Purpose

Hooks and advice represent the fifth most visible and frequently used extension point in the BI Dashboard Framework, enabling developers to create custom visualizations, interactive widgets, data displays, and user interface elements that seamlessly integrate with the dashboard ecosystem. This extension point is fundamental to the framework's value proposition—while the core provides essential charts (line, bar, pie, scatter) and standard widgets (tables, cards, metrics), the real power emerges when organizations can build domain-specific visualizations tailored to their unique business needs.

Component extensions can range from simple data displays (KPI cards, progress bars, status indicators) to sophisticated interactive visualizations (network graphs, geospatial maps, custom D3.js charts, WebGL-accelerated renderings) to complex composite widgets (filterable data tables, drill-down hierarchies, multi-step forms).

The framework supports multiple component authoring approaches—React components for developers familiar with the ecosystem, Web Components for framework-agnostic reusability, declarative configurations for simple cases, and render functions for lightweight elements—ensuring that component creation is accessible regardless of technical background or architectural preferences.

All component extensions benefit from automatic integration with the framework's state management, event system, theme engine, and layout manager, meaning developers can focus on visualization logic rather than infrastructure concerns. Components receive reactive data updates, respond to theme changes, participate in the command palette, and can be drag-dropped onto dashboards through the visual builder, providing a first-class user experience identical to built-in components.

3.7.2 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.

3.7.3 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

3.7.4 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

3.7.5 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

3.7.6 Architecture Patterns

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

```
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;
  };
}
```

3.7.7 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

Library	Type	Pros	Cons	Bundle Size	Use Case
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
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

3.7.8 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

Pattern	Description	Pros	Cons
Async Series	Sequential with results	Ordered; Result aggregation	Slower; Blocking

3.7.9 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

3.7.10 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

Platform	Hook System	Advice System	Implementation Details
Webpack	Tapable hooks	Compilation hooks	• Compiler hooks; • Compilation hooks; • Module hooks; • Asset optimization

3.7.11 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();
  }
}
```

3.7.12 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;
});
```

3.7.13 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

3.7.14 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

3.7.15 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

3.7.16 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

3.7.17 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.

3.8 Hot Reloading

3.8.1 Overview

Hot Module Replacement (HMR) enables developers to update extensions in real-time without losing application state, dramatically improving development velocity and the overall developer experience in the BI Dashboard Framework. HMR

represents a critical component of the "Developer-First Design" philosophy, transforming the traditional edit-save-refresh-navigate workflow into an instantaneous feedback loop where code changes appear immediately in the running application while preserving the current application state, user interactions, and data context.

This capability is particularly valuable in dashboard development scenarios where developers need to iterate on visualizations, tweak styling, adjust data transformations, or debug complex interactions—activities that traditionally required full page reloads, losing the current dashboard state, query results, filter selections, and navigation context.

The framework's HMR implementation goes beyond simple module replacement by intelligently handling extension lifecycle management: when an extension's code changes, the system gracefully unloads the old version (cleaning up event listeners, timers, and resources), loads the new version, restores relevant state (dashboard configuration, user preferences, cached data), and re-mounts components in their previous positions with their previous props.

This sophisticated orchestration ensures that developers can modify extension code, see changes instantly, and continue working from exactly where they left off without manually recreating their testing scenario. The HMR system also provides robust error handling—if a code change introduces a syntax error or runtime exception, the framework displays a helpful error overlay with stack traces and source maps, allows developers to fix the issue, and automatically recovers without requiring a manual page refresh.

This combination of speed, state preservation, and error resilience creates a development experience that feels more like live coding than traditional web development, enabling rapid experimentation and iteration that would be impractical with conventional reload-based workflows.

3.8.2 Core Capabilities

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

3.8.3 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

3.8.4 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

3.8.5 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
Hybrid	Critical state persisted	Balanced; Optimized	More logic

3.8.6 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

3.8.7 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

3.8.8 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();
  }
});
```

3.8.9 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.

3.9 Live Evaluation of DSL & JavaScript

3.9.1 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. This capability represents the convergence of the framework's extensibility philosophy with modern development practices, creating an Observable-style notebook experience where code cells can be written in either the custom DSL or full JavaScript, evaluated instantly, and automatically re-executed when their dependencies change.

Live evaluation transforms the dashboard from a static configuration into a dynamic, programmable environment where developers and data analysts can experiment with queries, test visualizations, prototype transformations, and explore data interactively without the friction of traditional compile-deploy-test cycles. The system maintains a dependency graph between cells—when a cell's output changes, all dependent cells automatically re-evaluate in topological order, ensuring consistency while minimizing unnecessary computation.

This reactive evaluation model, combined with hot module replacement for code changes and intelligent state preservation across reloads, creates a development experience that feels immediate and responsive. The framework's dual-language support means that simple dashboards can be built entirely in the declarative DSL for safety and simplicity, while complex scenarios can leverage JavaScript's full power for custom logic, external API integration, or advanced data processing.

Both evaluation paths are sandboxed for security, versioned for compatibility, and integrated with the same state management, event system, and component registry, ensuring a consistent experience regardless of which language is used. This live evaluation capability is particularly powerful for data exploration workflows—analysts can write SQL queries, transform results with JavaScript, visualize data with DSL-defined charts, and iterate on all three in real-time, seeing results update instantly as they refine their analysis.

3.9.2 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

3.9.3 JavaScript Live Evaluation

3.9.3.1 Architecture Pattern

Observable-style reactive cells with sandboxed execution. This pattern implements a cell-based evaluation model inspired by Observable notebooks and Jupyter, where each JavaScript code block is treated as an independent computational unit (cell) with explicit dependencies on other cells or data sources. The architecture maintains a directed acyclic graph (DAG) of cell dependencies, automatically tracking which cells reference which variables or outputs, and orchestrating re-evaluation in the correct order when upstream cells change.

Each cell executes in a sandboxed environment with controlled access to browser APIs, framework services, and other cells' outputs, preventing malicious code from compromising the application while still allowing legitimate use cases like data fetching, DOM manipulation, and state updates. The system supports both synchronous and asynchronous cell evaluation, handles circular dependency detection, provides detailed error reporting with source maps, and integrates with the HMR system to preserve cell outputs across code changes.

This architecture enables powerful interactive programming capabilities—cells can define functions used by other cells, fetch data from APIs, perform complex transformations, generate visualizations, or even dynamically register new components—all while maintaining security boundaries and providing instant feedback as code changes.

3.9.3.2 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());
        }
      });
    }
  }
}
```

3.9.3.3 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);
  }
}
```

3.9.4 DSL Live Compilation

3.9.4.1 Architecture Pattern

Compile-to-React with Vite HMR integration. This pattern treats DSL code as a high-level declarative specification that compiles down to optimized React components at build time or runtime, leveraging Vite's fast HMR capabilities to provide instant feedback during development.

The compilation pipeline parses DSL syntax into an abstract syntax tree (AST), validates the structure against the DSL schema, transforms the AST into React component definitions with proper hooks and lifecycle methods, and generates TypeScript-typed components that integrate seamlessly with the framework's component registry.

Unlike traditional template languages that require runtime interpretation, this compile-to-React approach produces first-class React components that benefit from React's optimization strategies (memoization, virtual DOM diffing, concurrent rendering) while maintaining the simplicity and safety of the DSL syntax. The Vite HMR integration enables incremental

compilation—when DSL code changes, only the affected components are recompiled and hot-swapped into the running application, preserving component state and avoiding full page reloads.

This architecture provides the best of both worlds: the developer-friendly syntax and safety guarantees of a DSL, combined with the performance and ecosystem benefits of React. The compiler also generates source maps that map DSL line numbers to React component code, enabling proper debugging and error reporting that references the original DSL source rather than the compiled output.

3.9.4.2 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();
  }
}
```

3.9.5 Hybrid Cell-Based System

3.9.5.1 Recommended Architecture

Combines DSL and JavaScript evaluation with unified state management into a cohesive hybrid system that supports both languages within the same dashboard, sharing state, data, and components seamlessly. This architecture recognizes that different parts of a dashboard have different requirements—some are best expressed declaratively in DSL (layouts, standard charts, simple filters), while others need JavaScript's full power (custom algorithms, API integrations, complex transformations)—and provides a unified execution environment where both can coexist and interoperate.

The system maintains a single dependency graph that tracks relationships between DSL cells, JavaScript cells, and shared data sources, ensuring that changes propagate correctly regardless of which language produced the output. State management is centralized through the framework's Zustand store, allowing DSL components to read and write the same

state that JavaScript cells manipulate, creating a truly integrated experience. The hybrid engine handles the complexity of coordinating two different evaluation models (compile-time DSL compilation vs.

runtime JavaScript interpretation), managing their different security profiles (DSL is inherently safe, JavaScript requires sandboxing), and ensuring consistent behavior across both. This architecture enables powerful composition patterns—a JavaScript cell can fetch data from an API, a DSL cell can visualize that data, another JavaScript cell can process user interactions, and a DSL cell can display the results—all working together as a cohesive unit with automatic dependency tracking and reactive updates.

3.9.5.2 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);
}
```

3.9.6 Performance Optimization

3.9.6.1 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
```

3.9.6.2 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);
  }
}
```

3.9.6.3 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>
  );
}
```

3.9.7 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

3.9.8 Usage Examples

3.9.8.1 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"
  }
}
```

3.9.8.2 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
    }))} />
```

```
  )
});
```

3.9.8.3 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} />);'
    }
  ]
};
```

3.9.8.4 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;
  }
}
```

Chapter 4

Security Model

4.1 Overview

A comprehensive security model protects users from malicious code execution while enabling powerful customization capabilities in an infinite canvas dashboard environment. The security model addresses the unique challenges of allowing users to write and execute custom JavaScript and DSL code within dashboard cells alongside trusted built-in components like tables, charts, inputs, and code editors.

The architecture employs a defense-in-depth strategy with multiple layers: sandboxed execution environments that isolate user code in separate iframes with their own origins, proxy-based API control that restricts built-in components to only their required permissions, runtime validation that checks every API call, resource limits that prevent runaway code from consuming excessive CPU or memory, audit logging for debugging and compliance, and rate limiting to prevent abuse.

This multi-layered approach recognizes that no single security mechanism is perfect—iframe sandboxing provides strong isolation but has communication overhead, proxy-based controls are performant but can be bypassed, and resource limits may need tuning—so the framework employs multiple overlapping defenses to ensure that even if one layer fails, others provide protection.

The security model is designed to be transparent to legitimate users: they can write normal JavaScript or custom DSL without learning complex security models, console logging works as expected, and the provided API surface is intuitive. Meanwhile, malicious code encounters multiple barriers: iframe origin isolation prevents access to the parent window, message validation blocks unauthorized communication, CSP policies restrict network access, and timeout mechanisms terminate long-running code.

This comprehensive approach allows the infinite canvas dashboard to support dynamic, user-generated code execution while maintaining the security, stability, and performance guarantees that production applications demand. The framework provides developer tooling—execution sandboxes for testing code safely, permission documentation for understanding API access, and clear error messages for security violations—ensuring that security becomes an enabler rather than an obstacle to building powerful, interactive dashboards.

4.2 Execution Environment

4.2.1 Recommended Architecture for Infinite Canvas Dashboard

For an infinite canvas dashboard with dynamic cells (tables, charts, inputs, CodeMirror editors) and on-the-fly code execution without page reloads, we recommend a **hybrid approach combining iframe Sandbox + Proxy-Based sandboxing**.

4.2.1.1 Architecture Decision

- **Primary: iframe Sandbox** (for custom code execution)

- **Why it's ideal:**
  - **Strong Isolation**: Custom JS/DSL code runs in complete isolation from main canvas
  - **No Page Reload**: iframes can be dynamically created/destroyed without affecting the main page
  - **Separate Crash Domain**: If user code crashes, it won't break the entire dashboard
  - **CSP Support**: Enforce strict Content Security Policies per cell
  - **True Sandboxing**: Each cell gets its own origin and execution context

- **Secondary: Proxy-Based** (for built-in components)
  - **Why it complements well:**
    - **Performance**: Built-in components (tables, charts, inputs) run in main context with controlled API access
    - **Fine-Grained Control**: Audit and limit what APIs each component type can access
    - **Flexibility**: Easy to add new component types without heavy isolation overhead
    - **Low Overhead**: No serialization cost for trusted components
  - **Direct DOM Access**: Components can manipulate DOM efficiently

4.2.1.2 Why NOT Other Approaches?

- **SES (Secure ECMAScript) - Not Suitable**
  - **Too restrictive**: No DOM access means you can't render UI components
  - **Learning curve**: Users would need to learn capability-based security
  - **Compatibility**: Many libraries won't work in frozen realm
  - **No visual output**: Can't create charts, tables, or interactive elements

- **Web Workers (alone) - Not Suitable**
  - **No DOM access**: Can't render tables, charts, or UI elements
  - **Message passing overhead**: Serialization cost for every UI update
  - **Limited use**: Only good for background computation (can be used as Layer 3)

- **VM Isolation - Not Suitable**
  - **Too heavy**: Significant overhead (1MB+) for each cell
  - **Limited browser support**: QuickJS-WASM not widely supported
  - **Overkill**: Don't need process-level isolation for dashboard cells
  - **Performance**: Slow startup time for each cell

4.2.1.3 Implementation Strategy

```
// Cell execution strategy
class CellExecutor {
  private iframePool: Map<string, HTMLIFrameElement> = new Map();
  private computeWorker: Worker;

  constructor() {
    // Initialize compute worker for heavy operations
    this.computeWorker = new Worker('compute-worker.js');
  }

  executeCell(cell: Cell): Promise<CellResult> {
    switch (cell.type) {
      case 'table':
      case 'chart':
      case 'input':
```

```
    case 'codemirror':
      // Built-in components: Proxy-based sandboxing
      return this.executeBuiltInComponent(cell);

    case 'custom-js':
    case 'custom-dsl':
      // User code: iframe sandboxing
      return this.executeUserCode(cell);

    case 'computation':
      // Heavy computation: Web Worker
      return this.executeComputation(cell);
  }
}

// Layer 1: Proxy-based for built-in components
private executeBuiltInComponent(cell: Cell): Promise<CellResult> {
  const allowedAPIs = this.getComponentPermissions(cell.type);

  const sandboxedAPI = new Proxy(this.dashboardAPI, {
    get(target, prop) {
      // Whitelist check
      if (!allowedAPIs.includes(prop as string)) {
        console.warn(`Component ${cell.type} attempted to access ${String(prop)}`);
        throw new Error(`API ${String(prop)} not allowed for ${cell.type}`);
      }

      // Audit logging
      this.auditLog({
        cellId: cell.id,
        cellType: cell.type,
        api: String(prop),
        timestamp: Date.now()
      });

      return target[prop];
    },

    set(target, prop, value) {
      throw new Error('Cannot modify dashboard API');
    }
  });

  // Execute in main context with controlled API
  const component = this.componentRegistry.get(cell.type);
  return component.render(cell.config, sandboxedAPI);
}

// Layer 2: iframe-based for user code
private async executeUserCode(cell: Cell): Promise<CellResult> {
  // Reuse iframe if possible
```

```
    let iframe = this.iframePool.get(cell.id);

    if (!iframe) {
      iframe = document.createElement('iframe');
      iframe.sandbox = 'allow-scripts'; // Minimal permissions
      iframe.style.cssText = `
        position: absolute;
        border: none;
        background: white;
      `;

      // Inject execution environment
      iframe.srcdoc = this.createSandboxHTML(cell);

      // Position in infinite canvas
      this.positionCellInCanvas(iframe, cell.position);

      // Add to DOM and pool
      document.getElementById('canvas-container').appendChild(iframe);
      this.iframePool.set(cell.id, iframe);

      // Wait for iframe to be ready
      await this.waitForIframeReady(iframe);
    }

    // Execute code in iframe
    return new Promise((resolve, reject) => {
      const messageHandler = (event: MessageEvent) => {
        if (event.source !== iframe.contentWindow) return;

        if (event.data.type === 'result' && event.data.cellId === cell.id) {
          window.removeEventListener('message', messageHandler);

          if (event.data.success) {
            resolve({
              output: event.data.output,
              logs: event.data.logs,
              executionTime: event.data.executionTime
            });
          } else {
            reject(new Error(event.data.error));
          }
        }
      };

      window.addEventListener('message', messageHandler);

      // Send code to execute
      iframe.contentWindow.postMessage({
        type: 'execute',
        cellId: cell.id,
```

```
      code: cell.code,
      language: cell.language,
      context: this.serializeContext(cell)
    }, '*');

    // Timeout after 30 seconds
    setTimeout(() => {
      window.removeEventListener('message', messageHandler);
      reject(new Error('Execution timeout'));
    }, 30000);
  });
}

// Layer 3: Web Worker for heavy computation
private executeComputation(cell: Cell): Promise<CellResult> {
  return new Promise((resolve, reject) => {
    const messageHandler = (event: MessageEvent) => {
      if (event.data.cellId === cell.id) {
        this.computeWorker.removeEventListener('message', messageHandler);
        resolve(event.data.result);
      }
    };

    this.computeWorker.addEventListener('message', messageHandler);
    this.computeWorker.postMessage({
      type: 'compute',
      cellId: cell.id,
      data: cell.data,
      operation: cell.operation
    });
  });
}

private createSandboxHTML(cell: Cell): string {
  return `
    <!DOCTYPE html>
    <html>
      <head>
        <meta charset="UTF-8">
        <style>
          body {
            margin: 0;
            padding: 16px;
            font-family: system-ui, -apple-system, sans-serif;
          }
          #output { min-height: 100px; }
          .error { color: red; padding: 8px; background: #fee; }
        </style>
      </head>
      <body>
        <div id="output"></div>
```

```
<script>
  const logs = [];
  const startTime = performance.now();

  // Override console for logging
  const originalConsole = { ...console };
  console.log = (...args) => {
    logs.push({ type: 'log', args });
    originalConsole.log(...args);
  };
  console.error = (...args) => {
    logs.push({ type: 'error', args });
    originalConsole.error(...args);
  };

  // Provide limited API to user code
  const api = {
    render: (html) => {
      document.getElementById('output').innerHTML = html;
    },
    createElement: (tag, props, children) => {
      const el = document.createElement(tag);
      if (props) Object.assign(el, props);
      if (children) el.innerHTML = children;
      return el;
    },
    appendTo: (element, selector = '#output') => {
      document.querySelector(selector).appendChild(element);
    }
  };

  // Listen for execution requests
  window.addEventListener('message', async (event) => {
    if (event.data.type === 'execute') {
      const { cellId, code, language, context } = event.data;

      try {
        let result;

        if (language === 'javascript') {
          // Execute JavaScript
          const fn = new Function('api', 'context', code);
          result = await fn(api, context);
        } else if (language === 'dsl') {
          // Execute custom DSL (implement your DSL parser)
          result = await this.executeDSL(code, api, context);
        }

        const executionTime = performance.now() - startTime;

        // Send result back
```

```
                window.parent.postMessage({
                  type: 'result',
                  cellId,
                  success: true,
                  output: result,
                  logs,
                  executionTime
                }, '*');

              } catch (error) {
                const errorDiv = document.createElement('div');
                errorDiv.className = 'error';
                errorDiv.textContent = error.message;
                document.getElementById('output').appendChild(errorDiv);

                window.parent.postMessage({
                  type: 'result',
                  cellId,
                  success: false,
                  error: error.message,
                  stack: error.stack,
                  logs
                }, '*');
              }
            }
          });

          // Signal ready
          window.parent.postMessage({ type: 'ready' }, '*');
        </script>
      </body>
    </html>
  `;
}

private getComponentPermissions(type: string): string[] {
  const permissions = {
    table: ['getData', 'formatData', 'exportCSV'],
    chart: ['getData', 'renderChart', 'updateChart'],
    input: ['getValue', 'setValue', 'validate'],
    codemirror: ['getValue', 'setValue', 'setLanguage', 'getSelection']
  };
  return permissions[type] || [];
}

private positionCellInCanvas(iframe: HTMLIFrameElement, position: Position) {
  iframe.style.left = `${position.x}px`;
  iframe.style.top = `${position.y}px`;
  iframe.style.width = `${position.width}px`;
  iframe.style.height = `${position.height}px`;
}
```

```
  private waitForIframeReady(iframe: HTMLIFrameElement): Promise<void> {
    return new Promise((resolve) => {
      const handler = (event: MessageEvent) => {
        if (event.source === iframe.contentWindow && event.data.type === 'ready') {
          window.removeEventListener('message', handler);
          resolve();
        }
      };
      window.addEventListener('message', handler);
    });
  }

  private serializeContext(cell: Cell): any {
    // Serialize only safe data for the cell context
    return {
      cellId: cell.id,
      variables: cell.variables,
      previousOutput: cell.previousOutput,
      // Don't serialize functions or DOM references
    };
  }

  private auditLog(entry: AuditLogEntry) {
    // Send to audit logging system
    console.log('[AUDIT]', entry);
  }

  // Cleanup when cell is removed
  destroyCell(cellId: string) {
    const iframe = this.iframePool.get(cellId);
    if (iframe) {
      iframe.remove();
      this.iframePool.delete(cellId);
    }
  }
}

// Usage example
const executor = new CellExecutor();

// Execute a chart cell (built-in, proxy-based)
await executor.executeCell({
  id: 'cell-1',
  type: 'chart',
  config: {
    data: [...],
    chartType: 'bar'
  }
});
```

```
// Execute custom JavaScript (iframe-based)
await executor.executeCell({
  id: 'cell-2',
  type: 'custom-js',
  language: 'javascript',
  code: `
    // User's custom code
    const data = context.previousOutput;
    const chart = api.createElement('div', { className: 'chart' });
    chart.innerHTML = '<h3>Custom Chart</h3>';
    api.appendTo(chart);
    return { success: true };
  `
});

// Execute heavy computation (worker-based)
await executor.executeCell({
  id: 'cell-3',
  type: 'computation',
  operation: 'aggregate',
  data: largeDataset
});
```

4.2.1.4 Three-Layer Defense Strategy

```
// Layer 1: Built-in components (fast, proxy-based)
const builtInComponents = {
  table: {
    permissions: ['getData', 'formatData', 'exportCSV'],
    executor: ProxySandboxedComponent
  },
  chart: {
    permissions: ['getData', 'renderChart', 'updateChart'],
    executor: ProxySandboxedComponent
  },
  input: {
    permissions: ['getValue', 'setValue', 'validate'],
    executor: ProxySandboxedComponent
  },
  codemirror: {
    permissions: ['getValue', 'setValue', 'setLanguage'],
    executor: ProxySandboxedComponent
  }
};

// Layer 2: User code (isolated, iframe-based)
const userCodeExecutor = {
  javascript: IframeSandboxedExecutor,
  typescript: IframeSandboxedExecutor,
  dsl: IframeSandboxedExecutor
};
```

```
// Layer 3: Heavy computation (parallel, worker-based)
const computeWorker = new ComputeWorkerPool({
  maxWorkers: 4,
  timeout: 60000
});

// Cell execution decision tree
async function executeCell(cell: Cell): Promise<CellResult> {
  // Fast path: built-in components
  if (cell.type in builtInComponents) {
    const component = builtInComponents[cell.type];
    return component.executor.render(cell, component.permissions);
  }

  // Check if computation should be offloaded
  if (cell.requiresComputation && cell.dataSize > LARGE_THRESHOLD) {
    return computeWorker.execute(cell);
  }

  // User code: iframe isolation
  return userCodeExecutor[cell.language].execute(cell);
}
```

4.2.1.5 Benefits for Infinite Canvas Dashboard

1. **Performance**
   - Built-in components run at native speed in main context
   - No serialization overhead for trusted components
   - iframe pooling reduces creation overhead
   - Web Workers handle heavy computation without blocking UI
2. **Safety**
   - User code completely isolated in iframes
   - Each iframe has its own origin (null origin)
   - Proxy layer prevents unauthorized API access
   - Crash in one cell doesn't affect others
3. **No Reload Required**
   - All execution happens dynamically
   - iframes created/destroyed on demand
   - Canvas state preserved across cell executions
   - Hot module replacement possible
4. **Scalability**
   - Each cell is independent
   - Can have hundreds of cells on canvas
   - Lazy loading: only execute visible cells
   - iframe pooling reduces memory footprint
5. **Developer Experience**
   - Users write normal JavaScript/DSL

- No need to learn new security models
- Rich API provided through `api` object
- Console logging works as expected

6. **Audit Trail**
   - Proxy layer logs all API usage
   - Track which cells access what data
   - Security monitoring and debugging
   - Compliance and forensics
7. **Flexibility**
   - Easy to add new component types
   - Can adjust permissions per component
   - Support multiple languages (JS, TS, DSL)
   - Extensible architecture

4.2.1.6 Real-World Inspiration

This approach is similar to:

- **Observable**: Uses iframes for user code, built-in components in main context
- **Jupyter**: Kernel isolation with rich front-end components
- **Figma**: Plugin isolation with controlled API surface
- **VS Code**: Extension host process with message passing
- **CodeSandbox**: iframe-based sandboxing for live preview

4.2.1.7 Security Considerations

```
// Validate message origins
window.addEventListener('message', (event) => {
  // Only accept messages from our iframes
  const iframe = Array.from(this.iframePool.values())
    .find(f => f.contentWindow === event.source);

  if (!iframe) {
    console.warn('Message from unknown source', event.origin);
    return;
  }

  // Process message
  this.handleCellMessage(event.data);
});

// Rate limiting for cell execution
class RateLimiter {
  private executions: Map<string, number[]> = new Map();

  canExecute(cellId: string): boolean {
    const now = Date.now();
    const recent = this.executions.get(cellId) || [];

    // Remove executions older than 1 minute
    const filtered = recent.filter(time => now - time < 60000);
```

```
    // Allow max 10 executions per minute
    if (filtered.length >= 10) {
      return false;
    }

    filtered.push(now);
    this.executions.set(cellId, filtered);
    return true;
  }
}

// Resource limits
const CELL_LIMITS = {
  maxExecutionTime: 30000,  // 30 seconds
  maxMemory: 50 * 1024 * 1024,  // 50MB per cell
  maxOutputSize: 1024 * 1024,  // 1MB output
  maxConcurrentExecutions: 5
};
```

4.2.1.8 Performance Optimizations

```
// 1. iframe pooling
class IframePool {
  private pool: HTMLIFrameElement[] = [];
  private maxSize = 10;

  acquire(): HTMLIFrameElement {
    return this.pool.pop() || this.createIframe();
  }

  release(iframe: HTMLIFrameElement) {
    if (this.pool.length < this.maxSize) {
      // Reset iframe state
      iframe.srcdoc = '';
      this.pool.push(iframe);
    } else {
      iframe.remove();
    }
  }
}

// 2. Lazy execution - only execute visible cells
class VisibilityManager {
  private observer: IntersectionObserver;

  constructor(private executor: CellExecutor) {
    this.observer = new IntersectionObserver((entries) => {
      entries.forEach(entry => {
        if (entry.isIntersecting) {
          const cellId = entry.target.getAttribute('data-cell-id');
          this.executor.executeCell(this.getCellById(cellId));
```

```
        }
      });
    }, { threshold: 0.1 });
  }

  observe(cellElement: HTMLElement) {
    this.observer.observe(cellElement);
  }
}

// 3. Debounced execution for code editors
function createDebouncedExecutor(delay = 500) {
  let timeout: number;

  return (cell: Cell) => {
    clearTimeout(timeout);
    timeout = setTimeout(() => {
      executor.executeCell(cell);
    }, delay);
  };
}
```

4.3 Real-World Implementation Examples

The following examples demonstrate how leading platforms implement similar sandboxing architectures for their canvas-based and notebook-style applications.

4.3.1 BI Dashboard Security Comparison

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

4.3.2 Observable-Style Reactive Cells

Observable uses reactive cell execution with automatic dependency tracking and iframe isolation for untrusted code.

```
// Observable-inspired reactive cell system
class ReactiveCellSystem {
  private cells: Map<string, Cell> = new Map();
  private dependencies: Map<string, Set<string>> = new Map();
```

```
  private executor: CellExecutor;

  constructor() {
    this.executor = new CellExecutor();
  }

  // Define a cell with dependencies
  defineCell(id: string, code: string, dependencies: string[] = []) {
    const cell: Cell = {
      id,
      type: 'custom-js',
      language: 'javascript',
      code,
      dependencies,
      value: undefined,
      status: 'pending'
    };

    this.cells.set(id, cell);
    this.dependencies.set(id, new Set(dependencies));
    this.maybeExecute(id);
  }

  private async maybeExecute(cellId: string) {
    const cell = this.cells.get(cellId);
    if (!cell) return;

    // Check if all dependencies are resolved
    const deps = this.dependencies.get(cellId) || new Set();
    const allResolved = Array.from(deps).every(depId => {
      const depCell = this.cells.get(depId);
      return depCell && depCell.status === 'fulfilled';
    });

    if (!allResolved) return;

    // Gather dependency values
    const context = {};
    deps.forEach(depId => {
      const depCell = this.cells.get(depId);
      if (depCell) context[depId] = depCell.value;
    });

    // Execute cell
    cell.status = 'pending';
    try {
      const result = await this.executor.executeCell({ ...cell, context });
      cell.value = result.output;
      cell.status = 'fulfilled';
      this.triggerDependents(cellId);
    } catch (error) {
```

```
      cell.status = 'rejected';
      cell.error = error.message;
    }
  }

  private triggerDependents(cellId: string) {
    this.dependencies.forEach((deps, id) => {
      if (deps.has(cellId)) this.maybeExecute(id);
    });
  }
}
```

4.3.3 Jupyter-Style Kernel Architecture

Jupyter separates execution (kernel) from UI (frontend) with message-based communication.

```
// Jupyter-inspired kernel architecture
class ExecutionKernel {
  private worker: Worker;
  private messageQueue: Map<string, {resolve: Function, reject: Function}> = new Map();

  constructor() {
    this.worker = new Worker('execution-kernel.js');
    this.worker.addEventListener('message', this.handleMessage.bind(this));
  }

  async execute(code: string, cellId: string): Promise<ExecutionResult> {
    const messageId = `${cellId}-${Date.now()}`;

    return new Promise((resolve, reject) => {
      this.messageQueue.set(messageId, { resolve, reject });

      this.worker.postMessage({
        type: 'execute_request',
        messageId,
        cellId,
        code
      });

      setTimeout(() => {
        if (this.messageQueue.has(messageId)) {
          this.messageQueue.delete(messageId);
          reject(new Error('Execution timeout'));
        }
      }, 30000);
    });
  }

  private handleMessage(event: MessageEvent) {
    const { type, messageId, status, data, error } = event.data;

    if (type === 'execute_reply') {
```

```
      const handler = this.messageQueue.get(messageId);
      if (handler) {
        this.messageQueue.delete(messageId);
        status === 'ok' ? handler.resolve(data) : handler.reject(new Error(error));
      }
    }
  }
}
```

4.3.4 Figma-Style Plugin Isolation

Figma runs plugins in isolated contexts with a controlled API surface based on declared permissions.

```
// Figma-inspired plugin system
class PluginHost {
  private plugins: Map<string, PluginInstance> = new Map();

  loadPlugin(manifest: PluginManifest, code: string) {
    const iframe = document.createElement('iframe');
    iframe.sandbox = 'allow-scripts';

    const pluginAPI = this.createPluginAPI(manifest.id, manifest.permissions);

    iframe.srcdoc = `
      <!DOCTYPE html>
      <html>
        <body>
          <script>
            window.figma = ${JSON.stringify(pluginAPI)};
            ${code}
          </script>
        </body>
      </html>
    `;

    document.body.appendChild(iframe);
    this.plugins.set(manifest.id, { id: manifest.id, iframe, manifest, api: pluginAPI });
  }

  private createPluginAPI(pluginId: string, permissions: string[]) {
    const api: any = {
      ui: {
        show: () => this.showPluginUI(pluginId),
        postMessage: (msg: any) => this.sendToPlugin(pluginId, msg)
      }
    };

    if (permissions.includes('read:canvas')) {
      api.currentPage = {
        selection: () => this.getSelection(),
        findAll: (selector: string) => this.findNodes(selector)
      };
```

```
    }

    if (permissions.includes('write:canvas')) {
      api.createRectangle = () => this.createNode('rectangle');
    }

    if (permissions.includes('network')) {
      api.fetch = (url: string, options: any) => this.proxyFetch(pluginId, url, options);
    }

    return api;
  }
}
```

4.3.5 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

---

4.3.5.1 1. SES (Secure ECMAScript)

Overview:

SES is a hardened subset of JavaScript developed by Agoric that provides deterministic, secure execution by removing ambient authority and non-deterministic features. It creates a "frozen realm" where all built-in objects are immutable and access to global state is controlled.

How It Works:

- **Compartments**: Creates isolated execution contexts with their own global scope
- **Frozen Intrinsics**: All built-in prototypes (`Object.prototype`, `Array.prototype`, etc.) are frozen to prevent prototype pollution
- **Capability-Based Security**: Code only has access to what's explicitly passed to it
- **Deterministic Execution**: Removes `Date.now()`, `Math.random()`, and other non-deterministic APIs

Implementation Example:

```
import { lockdown, Compartment } from 'ses';

// Harden the JavaScript environment
lockdown();

// Create isolated compartment
const compartment = new Compartment({
  // Endowments: explicitly provided capabilities
  console: {
    log: (...args) => console.log('[Extension]', ...args)
  }
});

// Execute untrusted code
const result = compartment.evaluate(`
  // No access to global window, document, or fetch
  // Only has access to provided console
  console.log('Hello from sandbox');
  return 42;
`);
```

Pros:

- **Strong Isolation**: Complete separation from host environment
- **Deterministic**: Same input always produces same output (useful for testing)
- **Prototype Safety**: Immune to prototype pollution attacks
- **Memory Safety**: Prevents memory leaks across compartments
- **Standards-Based**: Built on TC39 proposals

Cons:

- **Limited APIs**: No access to DOM, timers, network by default
- **Learning Curve**: Capability-based security requires different thinking
- **Compatibility**: May break code expecting full JavaScript environment
- **Performance**: Additional overhead from frozen intrinsics
- **Ecosystem**: Limited third-party library support

Best For:

- Financial applications requiring deterministic execution
- Plugin systems with untrusted code
- Smart contracts and blockchain applications
- High-security environments (banking, healthcare)
- Multi-tenant SaaS platforms

Real-World Usage:

- **Agoric**: Blockchain smart contracts
- **MetaMask Snaps**: Browser extension plugins
- **Salesforce LWC**: Lightning Web Components security

4.3.5.2 2. iframe Sandbox

Overview:

Uses HTML5 sandboxed iframes to create a completely isolated browsing context with its own origin, DOM, and JavaScript environment. The `sandbox` attribute restricts capabilities like form submission, scripts, and top navigation.

How It Works:

- **Origin Isolation**: Creates unique opaque origin (null origin)
- **CSP Integration**: Content Security Policy for additional restrictions
- **Permission Model**: Granular control via sandbox flags
- **Message Passing**: `postMessage` API for secure communication

Implementation Example:

```
// Create sandboxed iframe
const iframe = document.createElement('iframe');
iframe.sandbox = 'allow-scripts'; // Minimal permissions
iframe.srcdoc = `
  <!DOCTYPE html>
  <html>
    <body>
      <script>
        // Isolated JavaScript context
        window.parent.postMessage({
          type: 'result',
          data: 'Hello from sandbox'
        }, '*');
      </script>
    </body>
  </html>
`;

// Listen for messages
window.addEventListener('message', (event) => {
  if (event.source === iframe.contentWindow) {
    console.log('Received:', event.data);
  }
});

document.body.appendChild(iframe);
```

Sandbox Flags:

- `allow-scripts`: Enable JavaScript execution
- `allow-same-origin`: Allow same-origin access (dangerous with allow-scripts)
- `allow-forms`: Allow form submission

- `allow-popups`: Allow opening new windows
- `allow-modals`: Allow `alert()`, `confirm()`, etc.
- `allow-top-navigation`: Allow navigating top frame

Pros:

- **True Isolation**: Separate origin and execution context
- **Browser Native**: No external libraries required
- **CSP Support**: Leverage Content Security Policy
- **Resource Isolation**: Separate memory, storage, cookies
- **Well-Tested**: Battle-tested browser security feature

Cons:

- **Communication Overhead**: `postMessage` adds latency
- **Performance**: Full document context is heavyweight
- **Complexity**: Managing lifecycle and communication
- **Layout Issues**: CSS and positioning challenges
- **Debugging**: Harder to debug across iframe boundary

Best For:

- Embedding untrusted third-party content
- User-generated HTML/CSS/JavaScript
- Widget systems (ads, embeds)
- Preview environments
- Multi-tenant dashboards with custom visualizations

Security Considerations:

```
// DANGEROUS: Never combine these flags
iframe.sandbox = 'allow-scripts allow-same-origin';
// This allows sandbox escape!

// SAFE: Minimal permissions
iframe.sandbox = 'allow-scripts';

// SAFE: Validate message origins
window.addEventListener('message', (event) => {
  if (event.origin !== 'https://trusted-domain.com') {
    return; // Ignore untrusted messages
  }
});
```

4.3.5.3 3. Web Workers

Overview:

Web Workers run JavaScript in background threads separate from the main UI thread. They have no access to the DOM and communicate via message passing, providing natural isolation for compute-heavy tasks.

How It Works:

- **Thread Isolation**: Runs in separate thread (not process)
- **No DOM Access**: Cannot access `window`, `document`, or DOM APIs

- **Message Passing**: Structured cloning for data transfer
- **Shared Memory**: Optional `SharedArrayBuffer` for advanced use cases

Implementation Example:

```
// Main thread
const worker = new Worker('worker.js');

worker.postMessage({
  type: 'calculate',
  data: [1, 2, 3, 4, 5]
});

worker.addEventListener('message', (event) => {
  console.log('Result:', event.data);
});

// worker.js
self.addEventListener('message', (event) => {
  const { type, data } = event.data;

  if (type === 'calculate') {
    const sum = data.reduce((a, b) => a + b, 0);
    self.postMessage({ result: sum });
  }
});
```

Types of Workers:

- **Dedicated Workers**: One-to-one with creating script
- **Shared Workers**: Shared across multiple scripts/tabs
- **Service Workers**: Network proxy for offline support

Pros:

- **No DOM Access**: Natural security boundary
- **Parallel Execution**: True multi-threading
- **Isolated Scope**: Separate global scope
- **Resource Limits**: Browser manages memory
- **Transferable Objects**: Efficient large data transfer

Cons:

- **No UI Manipulation**: Cannot update DOM directly
- **Message Passing Overhead**: Serialization costs
- **Limited APIs**: No access to many browser APIs
- **Debugging Complexity**: Separate debugging context
- **Cold Start**: Worker initialization time

Best For:

- CPU-intensive calculations (data processing, parsing)
- Background data synchronization
- Image/video processing
- Cryptographic operations

- Large dataset transformations
- Real-time data analysis

Advanced Pattern - Comlink:

```
// Using Comlink library for easier RPC
import * as Comlink from 'comlink';

// worker.js
const api = {
  async processData(data) {
    // Heavy computation
    return data.map(x => x * 2);
  }
};

Comlink.expose(api);

// Main thread
const worker = new Worker('worker.js');
const api = Comlink.wrap(worker);

const result = await api.processData([1, 2, 3]);
console.log(result); // [2, 4, 6]
```

4.3.5.4 4. Proxy-Based Sandboxing

Overview:

Uses JavaScript Proxy objects to intercept and control access to APIs, objects, and properties. Provides fine-grained control over what code can access and how it behaves.

How It Works:

- **Interception**: Proxy traps intercept property access, function calls, etc.
- **Whitelisting**: Only allow access to approved APIs
- **Auditing**: Log all API usage for security monitoring
- **Transformation**: Modify behavior of existing APIs

Implementation Example:

```
// Create sandboxed global object
function createSandbox(permissions) {
  const sandbox = {
    console: {
      log: (...args) => console.log('[Sandbox]', ...args)
    }
  };

  // Proxy to control access
  return new Proxy(sandbox, {
    get(target, prop) {
      // Whitelist check
      if (prop in target) {
```

```
        return target[prop];
      }

      // Deny access to dangerous APIs
      if (['eval', 'Function', 'XMLHttpRequest'].includes(prop)) {
        throw new Error(`Access to ${prop} denied`);
      }

      // Audit logging
      console.warn(`Access attempt: ${prop}`);
      return undefined;
    },

    set(target, prop, value) {
      // Prevent modification of sandbox
      throw new Error('Cannot modify sandbox');
    },

    has(target, prop) {
      // Control 'in' operator
      return prop in target;
    }
  });
}

// Execute code in sandbox
const sandbox = createSandbox(['console']);
const fn = new Function('sandbox', `
  with (sandbox) {
    console.log('Hello'); // Works
    fetch('https://evil.com'); // undefined
  }
`);

fn(sandbox);
```

Advanced Pattern - Membrane:

```
// Membrane pattern: wrap all objects transitively
function createMembrane(target) {
  const wrapped = new WeakMap();

  function wrap(obj) {
    if (typeof obj !== 'object' || obj === null) {
      return obj;
    }

    if (wrapped.has(obj)) {
      return wrapped.get(obj);
    }

    const proxy = new Proxy(obj, {
```

```
      get(target, prop) {
        const value = target[prop];
        // Recursively wrap returned values
        return wrap(value);
      }
    });

    wrapped.set(obj, proxy);
    return proxy;
  }

  return wrap(target);
}
```

Pros:

- **Fine-Grained Control**: Precise API access control
- **Flexible**: Can modify behavior dynamically
- **Auditable**: Log all access attempts
- **No External Dependencies**: Pure JavaScript
- **Gradual Adoption**: Can wrap existing code incrementally

Cons:

- **Performance Overhead**: Proxy traps add latency
- **Complex Implementation**: Easy to make mistakes
- **Bypassable**: Can be circumvented with careful code
- **Maintenance**: Requires keeping up with new APIs
- **Not True Isolation**: Shares same JavaScript realm

Best For:

- API usage monitoring and analytics
- Gradual security hardening
- Development/debugging tools
- Permission-based API access
- Legacy code that can't be fully sandboxed

Security Limitations:

```
//  Proxy-based sandboxing can be bypassed:

// 1. Prototype access
({}).__proto__.constructor.constructor('alert(1)')();

// 2. Error stack traces
try {
  null.f();
} catch (e) {
  e.stack; // May contain references to outer scope
}

// 3. Async timing attacks
// Use timing to infer information about blocked APIs
```

4.3.5.5 5. VM Isolation (Separate JavaScript VM)

Overview:

Creates a completely separate JavaScript virtual machine with its own heap, garbage collector, and execution context. Provides the strongest isolation but with significant overhead.

How It Works:

- **Process Isolation**: Separate OS process (in some implementations)
- **Memory Isolation**: Separate heap and garbage collector
- **Resource Limits**: CPU, memory, and time limits
- **API Bridging**: Explicit serialization boundary

Implementation Example (Node.js vm2):

```
// Note: vm2 is Node.js only, not browser
const { VM } = require('vm2');

const vm = new VM({
  timeout: 1000, // 1 second timeout
  sandbox: {
    // Explicitly provided globals
    console: {
      log: (...args) => console.log('[VM]', ...args)
    }
  },
  eval: false, // Disable eval
  wasm: false  // Disable WebAssembly
});

try {
  const result = vm.run(`
    // Completely isolated environment
    console.log('Hello from VM');

    // No access to require, process, etc.
    // require('fs'); // Would throw error

    42
  `);

  console.log('Result:', result);
} catch (err) {
  console.error('VM error:', err);
}
```

Browser Implementation (QuickJS-WASM):

```
import { getQuickJS } from 'quickjs-emscripten';

async function runInVM(code) {
```

```
  const QuickJS = await getQuickJS();
  const vm = QuickJS.newContext();

  try {
    // Create isolated VM
    const result = vm.evalCode(code);

    if (result.error) {
      const error = vm.dump(result.error);
      result.error.dispose();
      throw error;
    }

    const value = vm.dump(result.value);
    result.value.dispose();
    return value;
  } finally {
    vm.dispose();
  }
}

// Usage
const result = await runInVM('2 + 2');
console.log(result); // 4
```

Pros:

- **Complete Isolation**: Strongest security boundary
- **Resource Limits**: Enforce CPU, memory, time limits
- **Crash Isolation**: VM crash doesn't affect host
- **Multiple Engines**: Can use different JS engines
- **Deterministic**: Can control all non-determinism

Cons:

- **Large Overhead**: Significant memory and CPU cost
- **Complex Setup**: Requires VM management
- **Limited Browser Support**: Mostly Node.js solutions
- **Serialization Cost**: All data must cross boundary
- **Debugging Difficulty**: Hard to debug across VM boundary

Best For:

- Maximum security requirements (financial, healthcare)
- Multi-tenant serverless platforms
- Code execution services (online IDEs, playgrounds)
- Plugin systems with untrusted code
- Blockchain smart contract execution

Resource Limiting:

```
const { VM } = require('vm2');
```

```javascript
const vm = new VM({
  timeout: 5000,        // 5 second timeout
  sandbox: {},
  fixAsync: true,       // Fix async timing issues

  // Custom console with rate limiting
  console: 'redirect',

  // Compiler options
  compiler: 'javascript',

  // Require options
  require: {
    external: false,    // No external modules
    builtin: [],        // No builtin modules
    root: './',
    mock: {}
  }
});

// Catch timeout errors
try {
  vm.run('while(true) {}'); // Will timeout
} catch (err) {
  console.error('Timeout:', err.message);
}
```

4.3.6 Comparison Matrix

Feature	SES	iframe	Web Worker	Proxy	VM
Isolation Strength	Strong	Very Strong	Strong	Weak	Very Strong
Performance	Good	Moderate	Good	Good	Poor
DOM Access	No	Yes (sandboxed)	No	Yes	No
Setup Complexity	Moderate	Low	Low	High	High
Browser Support	Modern	All	All	All	Limited
Memory Overhead	Low	High	Moderate	Low	Very High
Debugging	Moderate	Hard	Hard	Easy	Very Hard
Escape Risk	Very Low	Very Low	Low	High	Very Low

4.3.7 Hybrid Approach Recommendation

For a production BI dashboard extension system, consider combining multiple approaches:

```
// Layer 1: iframe for strong isolation
const extensionFrame = document.createElement('iframe');
extensionFrame.sandbox = 'allow-scripts';

// Layer 2: SES inside iframe for additional hardening
extensionFrame.srcdoc = `
  <script src="ses.umd.js"></script>
  <script>
    lockdown();

    const compartment = new Compartment({
      // Layer 3: Proxy-based API control
      dashboard: new Proxy(dashboardAPI, {
        get(target, prop) {
          // Audit and control access
          auditLog('API access:', prop);
          return target[prop];
        }
      })
    });

    // Layer 4: Web Worker for heavy computation
    const worker = new Worker('compute-worker.js');

    // Execute extension code
    compartment.evaluate(extensionCode);
  </script>
`;
```

This defense-in-depth approach provides:

- **iframe**: Process-level isolation
- **SES**: Capability-based security
- **Proxy**: Fine-grained API control and auditing
- **Web Worker**: Parallel execution without DOM access

4.4 Security Considerations

4.4.1 Message Origin Validation

Always validate the origin of messages from iframes to prevent unauthorized communication.

```
class SecureMessageBus {
  private iframeRegistry: Map<string, HTMLIFrameElement> = new Map();

  constructor() {
    window.addEventListener('message', this.handleMessage.bind(this));
  }

  private handleMessage(event: MessageEvent) {
    // Step 1: Validate origin (sandboxed iframes have "null" origin)
```

```
    if (event.origin !== 'null' && !this.isAllowedOrigin(event.origin)) {
      console.warn('[Security] Unauthorized origin:', event.origin);
      return;
    }

    // Step 2: Validate source
    const iframe = this.findIframeBySource(event.source);
    if (!iframe) {
      console.warn('[Security] Unknown iframe source');
      return;
    }

    // Step 3: Validate message structure
    if (!this.isValidMessage(event.data)) {
      console.warn('[Security] Invalid message structure');
      return;
    }

    // Step 4: Rate limiting
    if (!this.checkMessageRateLimit(iframe.cellId)) {
      console.warn('[Security] Rate limit exceeded');
      return;
    }

    // Step 5: Process message
    this.processMessage(iframe.cellId, event.data);
  }

  private findIframeBySource(source: WindowProxy) {
    for (const [cellId, iframe] of this.iframeRegistry) {
      if (iframe.contentWindow === source) {
        return { cellId, iframe };
      }
    }
    return null;
  }

  private isValidMessage(data: any): boolean {
    return (
      data &&
      typeof data === 'object' &&
      typeof data.type === 'string' &&
      ['result', 'error', 'log', 'ready'].includes(data.type)
    );
  }
}
```

4.4.2 Content Security Policy (CSP)

Implement strict CSP for iframe sandboxes to prevent XSS and data exfiltration.

```
class SecureIframeFactory {
  createSandboxedIframe(cellId: string, options: IframeOptions = {}): HTMLIFrameElement {
    const iframe = document.createElement('iframe');

    // Minimal sandbox permissions
    iframe.sandbox.add('allow-scripts');

    // NEVER add both allow-scripts and allow-same-origin together!
    // This would allow sandbox escape

    // Set strict CSP via meta tag
    const csp = this.buildCSP(options);

    iframe.srcdoc = `
      <!DOCTYPE html>
      <html>
        <head>
          <meta charset="UTF-8">
          <meta http-equiv="Content-Security-Policy" content="${csp}">
        </head>
        <body>
          <div id="output"></div>
          <script>
            // Prevent common escape attempts
            delete window.parent;
            delete window.top;
            delete window.frameElement;

            // Execution environment code here
          </script>
        </body>
      </html>
    `;

    iframe.setAttribute('referrerpolicy', 'no-referrer');
    return iframe;
  }

  private buildCSP(options: IframeOptions): string {
    const directives = [
      "default-src 'none'",
      "script-src 'unsafe-inline' 'unsafe-eval'",
      "style-src 'unsafe-inline'",
      "img-src data: blob:",
      "connect-src 'none'", // Block all network requests by default
      "frame-src 'none'",
      "object-src 'none'"
    ];

    if (options.allowedDomains?.length) {
      directives[4] = `connect-src ${options.allowedDomains.join(' ')}`;
```

```
    }

    return directives.join('; ');
  }
}
```

4.4.3 Resource Limits and Monitoring

Implement resource limits to prevent denial-of-service attacks.

```
class ResourceMonitor {
  private cellResources: Map<string, CellResources> = new Map();

  private readonly LIMITS = {
    maxExecutionTime: 30000, // 30 seconds
    maxMemory: 50 * 1024 * 1024, // 50MB
    maxOutputSize: 1024 * 1024, // 1MB
    maxConcurrentCells: 10,
    maxMessagesPerSecond: 100
  };

  startMonitoring(cellId: string) {
    const resources: CellResources = {
      cellId,
      startTime: Date.now(),
      messageCount: 0,
      outputSize: 0
    };

    // Set execution timeout
    resources.timeoutId = setTimeout(() => {
      this.terminateCell(cellId, 'Execution timeout exceeded');
    }, this.LIMITS.maxExecutionTime);

    this.cellResources.set(cellId, resources);
  }

  recordMessage(cellId: string) {
    const resources = this.cellResources.get(cellId);
    if (!resources) return;

    resources.messageCount++;
    const messagesPerSecond = resources.messageCount /
      ((Date.now() - resources.startTime) / 1000);

    if (messagesPerSecond > this.LIMITS.maxMessagesPerSecond) {
      this.terminateCell(cellId, 'Message rate limit exceeded');
    }
  }

  recordOutput(cellId: string, output: string) {
    const resources = this.cellResources.get(cellId);
```

```
    if (!resources) return;

    resources.outputSize += output.length;
    if (resources.outputSize > this.LIMITS.maxOutputSize) {
      this.terminateCell(cellId, 'Output size limit exceeded');
    }
  }

  private terminateCell(cellId: string, reason: string) {
    console.warn(`[Security] Terminating cell ${cellId}: ${reason}`);
    const resources = this.cellResources.get(cellId);
    if (resources?.timeoutId) {
      clearTimeout(resources.timeoutId);
    }
    this.cellResources.delete(cellId);
  }
}
```

4.4.4 DOM Access Control

Provide controlled DOM access through a sandboxed API with permission-based restrictions.

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

  if (permissions.includes('ui:render')) {
    // Limited DOM access - only safe tags allowed
    allowedAPIs.createElement = (tag: string) => {
      const allowedTags = ['div', 'span', 'p', 'button', 'h1', 'h2', 'h3', 'ul', 'li', 'table', 'tr', 'td'];
      if (!allowedTags.includes(tag)) {
        throw new Error(`Tag ${tag} not allowed`);
      }
      return document.createElement(tag);
    };

    allowedAPIs.querySelector = (selector: string) => {
      // Only allow querying within cell's own container
      const cellContainer = document.querySelector(`[data-cell-id="${extensionId}"]`);
      return cellContainer?.querySelector(selector);
    };
  }

  if (permissions.includes('storage:local')) {
    // Namespaced storage to prevent conflicts
    allowedAPIs.storage = {
      get: (key: string) => localStorage.getItem(`ext_${extensionId}_${key}`),
      set: (key: string, value: string) => {
        // Limit storage size per extension
        const maxSize = 1024 * 1024; // 1MB
        if (value.length > maxSize) {
          throw new Error('Storage quota exceeded');
```

```
        }
        localStorage.setItem(`ext_${extensionId}_${key}`, value);
      },
      remove: (key: string) => localStorage.removeItem(`ext_${extensionId}_${key}`)
    };
  }

  if (permissions.includes('network:fetch')) {
    // Proxied fetch with URL whitelist
    allowedAPIs.fetch = async (url: string, options?: RequestInit) => {
      // Check against allowed domains
      const allowedDomains = ['api.example.com', 'data.example.com'];
      const urlObj = new URL(url);

      if (!allowedDomains.some(domain => urlObj.hostname.endsWith(domain))) {
        throw new Error(`Domain ${urlObj.hostname} not allowed`);
      }

      // Add rate limiting
      // Add audit logging
      return fetch(url, options);
    };
  }

  return allowedAPIs;
};
```

4.5 Performance Optimizations

4.5.1 iframe Pooling

Reuse iframes to reduce creation overhead and improve performance.

```
class IframePool {
  private available: HTMLIFrameElement[] = [];
  private inUse: Map<string, HTMLIFrameElement> = new Map();
  private readonly maxPoolSize = 10;

  acquire(cellId: string): HTMLIFrameElement {
    let iframe = this.available.pop();
    if (!iframe) {
      iframe = this.createIframe();
    }
    this.inUse.set(cellId, iframe);
    return iframe;
  }

  release(cellId: string) {
    const iframe = this.inUse.get(cellId);
    if (!iframe) return;
```

```
    this.inUse.delete(cellId);
    this.resetIframe(iframe);

    if (this.available.length < this.maxPoolSize) {
      this.available.push(iframe);
    } else {
      iframe.remove();
    }
  }

  private createIframe(): HTMLIFrameElement {
    const iframe = document.createElement('iframe');
    iframe.sandbox.add('allow-scripts');
    iframe.style.cssText = 'position: absolute; border: none; visibility: hidden;';
    document.body.appendChild(iframe);
    return iframe;
  }

  private resetIframe(iframe: HTMLIFrameElement) {
    iframe.srcdoc = '';
    iframe.style.visibility = 'hidden';
  }
}
```

4.5.2 Lazy Execution with Intersection Observer

Only execute cells that are visible in the viewport.

```
class LazyExecutionManager {
  private observer: IntersectionObserver;
  private pendingCells: Map<string, Cell> = new Map();

  constructor(private executor: CellExecutor) {
    this.observer = new IntersectionObserver(
      (entries) => this.handleIntersection(entries),
      {
        root: null,
        rootMargin: '100px', // Start loading 100px before visible
        threshold: 0.1
      }
    );
  }

  registerCell(cellElement: HTMLElement, cell: Cell) {
    cellElement.setAttribute('data-cell-id', cell.id);
    this.pendingCells.set(cell.id, cell);
    this.observer.observe(cellElement);
  }

  private handleIntersection(entries: IntersectionObserverEntry[]) {
    entries.forEach(entry => {
```

```
      if (entry.isIntersecting) {
        const cellId = entry.target.getAttribute('data-cell-id');
        if (!cellId) return;

        const cell = this.pendingCells.get(cellId);
        if (!cell) return;

        this.executeCell(cell, entry.target as HTMLElement);
        this.observer.unobserve(entry.target);
        this.pendingCells.delete(cellId);
      }
    });
  }

  private async executeCell(cell: Cell, element: HTMLElement) {
    element.classList.add('cell-loading');
    try {
      const result = await this.executor.executeCell(cell);
      element.classList.remove('cell-loading');
      element.classList.add('cell-success');
      this.renderOutput(element, result);
    } catch (error) {
      element.classList.add('cell-error');
      this.renderError(element, error);
    }
  }
}
```

4.5.3 Debounced Execution for Code Editors

Prevent excessive executions while user is typing.

```
class DebouncedExecutor {
  private timeouts: Map<string, number> = new Map();
  private readonly defaultDelay = 500; // ms

  constructor(private executor: CellExecutor) {}

  execute(cell: Cell, delay: number = this.defaultDelay): Promise<CellResult> {
    return new Promise((resolve, reject) => {
      const existingTimeout = this.timeouts.get(cell.id);
      if (existingTimeout) {
        clearTimeout(existingTimeout);
      }

      const timeoutId = setTimeout(async () => {
        this.timeouts.delete(cell.id);
        try {
          const result = await this.executor.executeCell(cell);
          resolve(result);
        } catch (error) {
          reject(error);
```

```
        }
      }, delay);

      this.timeouts.set(cell.id, timeoutId);
    });
  }
}

// Usage with CodeMirror
editor.on('change', () => {
  const code = editor.getValue();
  const cell: Cell = { id: 'editor-cell', type: 'custom-js', language: 'javascript', code };

  debouncedExecutor.execute(cell, 500)
    .then(result => updateOutput(result))
    .catch(error => showError(error));
});
```

4.6 Capability-Based Permissions

4.6.1 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"
  ]
}
```

4.6.2 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

Permission	Description	Risk Level	Use Cases
`network:websocket`	WebSocket connections	High	Real-time data
`system:commands`	Register commands	Low	Custom actions
`system:keybindings`	Register keybindings	Low	Keyboard shortcuts
`system:eval`	Execute arbitrary code	Critical	Advanced extensions (rarely granted)

4.6.3 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

4.6.4 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.

4.7 API Surface

4.7.1 Purpose

The API surface provides a carefully curated set of functions, hooks, and interfaces that define how extensions interact with the dashboard framework, serving as the contract between the core system and third-party code.

This API represents the framework's public interface—the stable, versioned, documented set of capabilities that extensions can depend on across releases—and is designed with several critical goals in mind: minimalism (exposing only necessary functionality to reduce attack surface and maintenance burden), type safety (comprehensive TypeScript definitions that catch errors at compile time rather than runtime), explicitness (clear, self-documenting function names and parameters that make code intent obvious), auditability (logging all sensitive operations for security monitoring and debugging), and stability (maintaining backward compatibility through semantic versioning and deprecation policies).

The API surface acts as a security boundary, mediating all extension access to core framework capabilities—component registration, state management, event subscription, command execution, data access, and UI manipulation—through well-defined entry points that enforce permissions, validate inputs, and log operations. By controlling what extensions can do through a limited API rather than exposing internal implementation details, the framework maintains flexibility to refactor internals, optimize performance, and fix bugs without breaking extensions.

The API is organized into logical namespaces (components, commands, state, events, storage, network) that group related functionality, uses consistent patterns across all endpoints (promise-based async operations, error handling conventions, naming schemes), and provides both low-level primitives for advanced use cases and high-level convenience functions for common patterns. This thoughtful API design ensures that extension developers have the power they need while the framework maintains the control it requires for security, stability, and evolution.

4.7.2 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

4.7.3 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

4.7.4 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.

4.8 Code Review & Signing

4.8.1 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

4.8.2 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
  );
}
```

4.8.3 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

4.8.4 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;
}
```

4.8.5 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 that elevate the framework beyond traditional static dashboards into a sophisticated, interactive data exploration and collaboration platform.

This section covers advanced architectural patterns and features that distinguish cutting-edge BI tools from conventional reporting systems:

Canvas Architecture for infinite, free-form workspace layouts inspired by tools like Count.co and tldraw, enabling users to arrange visualizations spatially rather than being constrained by rigid grid systems.

SQL Integration that treats SQL as a first-class language alongside the DSL and JavaScript, with live query execution, result caching, and intelligent query optimization.

Real-Time Collaboration enabling multiple users to work simultaneously on the same dashboard with operational transformation for conflict resolution, presence awareness showing who's viewing what, and live cursors tracking collaborator actions.

Performance Optimization strategies including virtualization for rendering only visible elements, memoization to prevent unnecessary re-renders, code splitting for faster initial loads, and Web Workers for offloading CPU-intensive computations.

Time-Travel Debugging that records application state over time, allowing developers to step backward through state changes to diagnose issues.

Offline Support through service workers and local-first architecture, ensuring dashboards remain functional without network connectivity.

Advanced Data Processing with DuckDB WASM for in-browser analytics on large datasets, streaming data support for real-time updates, and incremental computation for efficient reactive updates.

These advanced features transform the BI Dashboard Framework from a simple visualization tool into a comprehensive platform for data analysis, enabling workflows that were previously only possible in desktop applications or specialized tools.

Each feature is designed to integrate seamlessly with the core architecture—canvas interfaces work with the component registry, SQL cells participate in the dependency graph, collaboration syncs through the state management system, and performance optimizations respect the plugin lifecycle—ensuring that advanced capabilities enhance rather than complicate the developer experience.

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, fundamentally reimagining how users interact with dashboards by replacing traditional fixed-grid layouts with a boundless, zoomable canvas where visualizations, data tables, text annotations, and interactive elements can be positioned freely in two-dimensional space.

This canvas paradigm mirrors physical whiteboards and design tools like Figma or Miro, creating an intuitive spatial environment where users can organize information according to their mental models rather than conforming to predetermined layouts.

The infinite canvas architecture enables powerful workflows: users can zoom out to see the big picture of an entire analysis, zoom in to focus on specific details, pan across different sections of a dashboard, group related visualizations spatially, draw connections between data points, and create hierarchical layouts where high-level summaries link to detailed drill-downs positioned nearby.

This approach is particularly valuable for exploratory data analysis, where the structure of the analysis emerges organically as users investigate data, add visualizations, annotate findings, and reorganize elements to tell a coherent story.

The technical implementation leverages HTML5 Canvas or WebGL for high-performance rendering, viewport transformation matrices for smooth pan and zoom, spatial indexing (R-trees or quadtrees) for efficient collision detection and visibility queries, and virtualization to render only elements within the visible viewport, ensuring performance even with hundreds of visualization cells.

The canvas system integrates deeply with the framework's component architecture—any registered component can be instantiated as a canvas cell, cells can communicate through the event system, state changes trigger selective re-renders, and the entire canvas state serializes for persistence and collaboration. This canvas-first approach represents a significant evolution in BI dashboard design, moving from static report layouts to dynamic, explorable workspaces that adapt to user needs rather than imposing rigid structure.

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

System	Description	Use Case
Relative Positioning	Position relative to parent	Nested components, groups
Auto-Layout	Algorithm-based positioning	Automatic organization, graphs
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

Platform	Canvas Implementation	Key Features
Count.co	React + Canvas	• Cell-based layout; • Free-form positioning; • Auto-layout options; • SQL-driven cells
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

Library	Language	Pros	Cons	Bundle Size	Use Case
Automerge	JavaScript/Rust	Pure CRDT; Offline-first; Time travel	Larger bundle; Performance	~200KB	Offline-first apps
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'));
```

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

// 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

Comprehensive data persistence architecture for the BI Dashboard Framework, encompassing configuration management, user preferences, dashboard state, query results, cached data, and application metadata. Data Persistence represents one of the most critical architectural concerns in modern web applications, particularly for data-intensive BI tools where users expect their work to be preserved across sessions, synchronized across devices, and accessible even when offline.

This system must balance competing requirements: performance (fast reads and writes), reliability (no data loss), scalability (handle large datasets), security (encrypt sensitive data), and user experience (seamless synchronization, conflict resolution, offline support). The framework implements a multi-tiered persistence strategy that leverages browser storage APIs (localStorage, IndexedDB, Cache API), optional cloud backends (REST APIs, WebSocket for real-time sync), and intelligent caching layers to provide optimal performance while ensuring data durability.

Unlike traditional server-centric applications where persistence is straightforward—write to database, read from database—client-side web applications face unique challenges including storage quota limits (typically 50MB-1GB depending on browser), lack of transactions across storage types, browser-specific quirks and bugs, and the need to handle offline scenarios gracefully.

The Data Persistence architecture addresses these challenges through careful storage selection (localStorage for small key-value data, IndexedDB for structured data and large objects, Cache API for HTTP responses), intelligent quota management (compression, cleanup policies, user notifications), and robust error handling (fallbacks, retries, user-facing error messages). This section explores the complete persistence stack from low-level storage APIs through high-level abstractions like settings management and state synchronization, providing architectural guidance, implementation patterns, performance optimizations, and security best practices for building a production-grade BI dashboard framework.

6.1 Overview

6.1.1 Persistence Layers

The framework implements a three-tier persistence architecture, each layer serving distinct purposes with specific trade-offs:

1. Client-Side Persistence (Browser Storage)

- **Purpose**: Immediate access, offline support, reduced server load
- **Technologies**: localStorage, IndexedDB, Cache API, Service Workers
- **Use Cases**: User preferences, dashboard layouts, cached query results, offline data
- **Advantages**: Fast access, works offline, reduces network traffic
- **Limitations**: Storage quotas (50MB-1GB), browser-specific, no cross-device sync
- **Data Types**: Settings, UI state, cached responses, temporary data

2. Server-Side Persistence (Cloud Storage)

- **Purpose**: Cross-device sync, backup, collaboration, unlimited storage
- **Technologies**: PostgreSQL, MongoDB, S3, Redis
- **Use Cases**: Shared dashboards, user accounts, large datasets, audit logs
- **Advantages**: Unlimited storage, cross-device access, backup/recovery
- **Limitations**: Network latency, requires connectivity, server costs
- **Data Types**: User accounts, shared resources, historical data, backups

3. Hybrid Persistence (Offline-First with Sync)

- **Purpose**: Best of both worlds—offline capability with cloud backup
- **Technologies**: IndexedDB + REST API, Service Worker + Background Sync
- **Use Cases**: Collaborative dashboards, mobile access, unreliable networks
- **Advantages**: Works offline, syncs when online, conflict resolution
- **Limitations**: Complex implementation, sync conflicts, storage management
- **Data Types**: Dashboards, queries, annotations, user-generated content

6.1.2 Storage Technology Comparison

Technology	Capacity	API Type	Use Case	Persistence	Performance
localStorage	~5-10MB	Synchronous	Small key-value data	Permanent	Fast (in-memory)
sessionStorage	~5-10MB	Synchronous	Temporary session data	Session only	Fast (in-memory)
IndexedDB	~50MB-1GB+	Asynchronous	Structured data, large objects	Permanent	Fast (indexed queries)
Cache API	~50MB-1GB+	Asynchronous (Promise)	HTTP responses, assets	Permanent	Very fast (optimized for HTTP)
Web SQL	~50MB	Asynchronous (callback)	Deprecated	Permanent	Fast (SQL queries)
File System API	User-granted	Asynchronous (Promise)	Large files, user documents	Permanent	Fast (native FS)

6.1.3 Data Categories

The framework persists several distinct categories of data, each with different requirements:

1. Configuration Data

- User preferences (theme, language, timezone)
- Application settings (auto-save, notifications)
- Workspace configurations (panel layouts, shortcuts)
- **Size**: Small (1-100KB)
- **Update Frequency**: Low (occasional changes)
- **Storage**: localStorage or IndexedDB
- **Sync**: Optional cloud backup

2. Dashboard State

- Dashboard definitions (layouts, components, queries)

- Filter states and selections
- Drill-down paths and navigation history
- **Size**: Medium (100KB-10MB)
- **Update Frequency**: Medium (frequent edits)
- **Storage**: IndexedDB with versioning
- **Sync**: Required for collaboration

3. Query Results

- Cached data from database queries
- Aggregated metrics and calculations
- Time-series data for charts
- **Size**: Large (1MB-100MB+)
- **Update Frequency**: High (real-time updates)
- **Storage**: IndexedDB with TTL expiration
- **Sync**: Not synced (regenerated on-demand)

4. User-Generated Content

- Annotations and comments
- Custom calculations and formulas
- Saved filters and bookmarks
- **Size**: Small to Medium (1KB-1MB)
- **Update Frequency**: Low to Medium
- **Storage**: IndexedDB + server backup
- **Sync**: Required for sharing

5. Application Metadata

- Audit logs and usage analytics
- Error reports and diagnostics
- Performance metrics
- **Size**: Medium (100KB-10MB)
- **Update Frequency**: High (continuous logging)
- **Storage**: IndexedDB with rotation
- **Sync**: Periodic upload to server

6.1.4 Persistence Strategies

Choosing the right persistence strategy is critical for balancing data safety, performance, and user experience. The BI Dashboard Framework supports multiple persistence patterns, each optimized for different data types, update frequencies, and criticality levels. The strategy selection depends on several factors: how frequently the data changes (high-frequency updates benefit from debouncing), how critical the data is (user settings require immediate persistence), whether the operation is user-initiated or automatic (manual vs.

automatic saves), network availability (offline scenarios need local-first approaches), and performance constraints (large datasets may require background processing). Modern applications often employ a hybrid approach, using different strategies for different data categories—immediate persistence for critical settings, debounced persistence for UI state, periodic auto-save for documents, and optimistic updates for collaborative features. Understanding these patterns and their trade-offs enables developers to build robust persistence layers that feel responsive while ensuring data durability.

6.1.4.1 Strategy Comparison Matrix

Strategy	Data Safety	Performance	UX Impact	Complexity	Best Use Case
Immediate	Highest	Low (blocking)	Noticeable lag	Low	Critical settings, explicit saves
Debounced	High	High	Smooth	Medium	Frequently changing UI state
Periodic	Medium	High	Transparent	Low	Auto-save, draft preservation
Manual	Variable	Highest	Requires action	Low	Power-user tools, large documents
Optimistic	High	Highest	Seamless	High	Collaborative editing, real-time sync
Lazy	Low	Highest	Transparent	Medium	Non-critical cache, prefetch data
Transactional	Highest	Medium	Depends	High	Multi-step operations, data integrity

6.1.4.2 1. Immediate Persistence

Write data to storage synchronously or with minimal delay immediately after every change, ensuring maximum data safety at the cost of performance.

Characteristics

- Zero or minimal delay between change and persistence
- Blocks UI until write completes (if synchronous)
- Guarantees data is saved before next operation
- Simple error handling (fail immediately)

Implementation Patterns

```
// Pattern 1: Synchronous localStorage (simple, blocking)
function saveSettingImmediate(key: string, value: any) {
  try {
    localStorage.setItem(key, JSON.stringify(value));
  } catch (error) {
    handleStorageError(error);
  }
}

// Pattern 2: Immediate IndexedDB (async, non-blocking)
async function saveDashboardImmediate(dashboard: Dashboard) {
  const db = await openDB('dashboards');
  const tx = db.transaction('dashboards', 'readwrite');

  try {
    await tx.objectStore('dashboards').put(dashboard);
    await tx.done;
    showSuccessNotification('Dashboard saved');
  } catch (error) {
    showErrorNotification('Failed to save dashboard');
```

```
    throw error;
  }
}

// Pattern 3: Immediate with optimistic UI update
async function updateSettingImmediate(key: string, value: any) {
  // Update UI immediately
  updateUIState(key, value);

  try {
    // Persist to storage
    await persistToIndexedDB(key, value);
  } catch (error) {
    // Rollback UI on failure
    revertUIState(key);
    showErrorNotification('Failed to save setting');
  }
}
```

Use Cases

- User explicitly clicks "Save" button
- Critical settings (security preferences, data source credentials)
- One-time configuration changes
- Small data payloads (<10KB)
- When data loss is unacceptable

Advantages

- Maximum data safety—no risk of losing unsaved changes
- Simple mental model for users (save = persisted)
- Immediate feedback on storage errors
- No complex timing logic required

Disadvantages

- Performance overhead on every change
- Can cause UI lag if persistence is slow
- May trigger excessive writes for rapid changes
- Synchronous operations block the main thread

6.1.4.3 2. Debounced Persistence

Batch multiple rapid changes and persist only after a quiet period (e.g., 500ms-2s after last change), optimizing for performance while maintaining reasonable data safety.

Characteristics

- Delays persistence until user stops making changes
- Cancels pending writes when new changes occur
- Reduces write frequency for rapid successive updates
- Balances performance and data safety

Implementation Patterns

```typescript
// Pattern 1: Basic debounce with lodash
import { debounce } from 'lodash';

const debouncedSave = debounce(async (data: any) => {
  await persistToIndexedDB(data);
}, 1000); // Wait 1s after last change

function onFilterChange(filter: Filter) {
  updateUIState(filter);
  debouncedSave(filter);
}

// Pattern 2: Custom debounce with TypeScript
function createDebouncedPersistence<T>(
  persistFn: (data: T) => Promise<void>,
  delay: number = 1000
) {
  let timeoutId: number | null = null;
  let pendingData: T | null = null;

  return {
    save: (data: T) => {
      pendingData = data;

      if (timeoutId !== null) {
        clearTimeout(timeoutId);
      }

      timeoutId = window.setTimeout(async () => {
        if (pendingData !== null) {
          try {
            await persistFn(pendingData);
            pendingData = null;
          } catch (error) {
            console.error('Debounced save failed:', error);
          }
        }
        timeoutId = null;
      }, delay);
    },

    flush: async () => {
      if (timeoutId !== null) {
        clearTimeout(timeoutId);
        timeoutId = null;
      }
      if (pendingData !== null) {
        await persistFn(pendingData);
        pendingData = null;
      }
    },
```

```
    cancel: () => {
      if (timeoutId !== null) {
        clearTimeout(timeoutId);
        timeoutId = null;
      }
      pendingData = null;
    }
  };
}

// Usage
const dashboardPersistence = createDebouncedPersistence(
  async (dashboard: Dashboard) => {
    await saveDashboardToIndexedDB(dashboard);
  },
  1500
);

// Save changes with debouncing
function onDashboardEdit(dashboard: Dashboard) {
  updateUIState(dashboard);
  dashboardPersistence.save(dashboard);
}

// Force immediate save on navigation
window.addEventListener('beforeunload', () => {
  dashboardPersistence.flush();
});

// Pattern 3: Debounce with leading edge (save immediately, then debounce)
function createLeadingDebounce<T>(
  persistFn: (data: T) => Promise<void>,
  delay: number = 1000
) {
  let timeoutId: number | null = null;
  let lastSaveTime = 0;

  return async (data: T) => {
    const now = Date.now();
    const timeSinceLastSave = now - lastSaveTime;

    // Save immediately if enough time has passed
    if (timeSinceLastSave > delay) {
      lastSaveTime = now;
      await persistFn(data);
      return;
    }

    // Otherwise, debounce
    if (timeoutId !== null) {
```

```
      clearTimeout(timeoutId);
    }

    timeoutId = window.setTimeout(async () => {
      lastSaveTime = Date.now();
      await persistFn(data);
      timeoutId = null;
    }, delay);
  };
}
```

Use Cases

- Text input fields (search queries, filter values)
- Slider controls (adjusting chart parameters)
- Drag-and-drop positioning (panel layouts)
- Frequently changing UI state (scroll positions, zoom levels)
- Form fields with auto-save

Advantages

- Reduces write frequency (better performance)
- Smooth user experience (no lag on each keystroke)
- Automatically batches rapid changes
- Lower storage wear and tear

Disadvantages

- Potential data loss if browser crashes during delay
- Complexity in managing pending writes
- Requires careful handling of navigation/logout
- May confuse users about save state

Best Practices

- Use 500ms-2s delay depending on data criticality
- Flush pending writes on `beforeunload` event
- Show "Saving..." indicator during debounce period
- Provide manual "Save Now" option for user control
- Consider leading-edge debounce for immediate feedback

6.1.4.4 3. Periodic Persistence

Save data at fixed time intervals (e.g., every 30 seconds) regardless of whether changes occurred, providing predictable auto-save behavior.

Characteristics

- Saves on a fixed schedule (e.g., every 30s, 1min, 5min)
- Independent of user actions
- Predictable save points
- May save unchanged data (wasteful)

Implementation Patterns

```typescript
// Pattern 1: Simple interval-based auto-save
class AutoSaveManager {
  private intervalId: number | null = null;
  private isDirty = false;
  private currentData: any = null;

  start(interval: number = 30000) {
    this.intervalId = window.setInterval(async () => {
      if (this.isDirty && this.currentData) {
        await this.save();
      }
    }, interval);
  }

  markDirty(data: any) {
    this.isDirty = true;
    this.currentData = data;
  }

  private async save() {
    try {
      await persistToIndexedDB(this.currentData);
      this.isDirty = false;
      showAutoSaveIndicator('Saved at ' + new Date().toLocaleTimeString());
    } catch (error) {
      showAutoSaveIndicator('Auto-save failed');
    }
  }

  stop() {
    if (this.intervalId !== null) {
      clearInterval(this.intervalId);
      this.intervalId = null;
    }
  }

  async forceSave() {
    if (this.isDirty) {
      await this.save();
    }
  }
}

// Usage
const autoSave = new AutoSaveManager();
autoSave.start(30000); // Auto-save every 30 seconds

function onDocumentEdit(document: Document) {
  updateUIState(document);
  autoSave.markDirty(document);
}
```

```typescript
// Pattern 2: Adaptive interval (adjust based on activity)
class AdaptiveAutoSave {
  private intervalId: number | null = null;
  private baseInterval = 30000; // 30 seconds
  private activeInterval = 10000; // 10 seconds when active
  private inactiveInterval = 60000; // 1 minute when inactive
  private lastActivityTime = Date.now();
  private currentData: any = null;

  start() {
    this.updateInterval();
  }

  onActivity(data: any) {
    this.lastActivityTime = Date.now();
    this.currentData = data;
    this.updateInterval();
  }

  private updateInterval() {
    const timeSinceActivity = Date.now() - this.lastActivityTime;
    const interval = timeSinceActivity < 60000
      ? this.activeInterval
      : this.inactiveInterval;

    if (this.intervalId !== null) {
      clearInterval(this.intervalId);
    }

    this.intervalId = window.setInterval(async () => {
      if (this.currentData) {
        await persistToIndexedDB(this.currentData);
      }
      this.updateInterval(); // Re-evaluate interval
    }, interval);
  }

  stop() {
    if (this.intervalId !== null) {
      clearInterval(this.intervalId);
    }
  }
}

// Pattern 3: Periodic with version history
class VersionedAutoSave {
  private intervalId: number | null = null;
  private maxVersions = 10;

  start(interval: number = 30000) {
```

```
    this.intervalId = window.setInterval(async () => {
      await this.saveVersion();
    }, interval);
  }

  private async saveVersion() {
    const currentState = getCurrentDashboardState();
    const version = {
      timestamp: Date.now(),
      data: currentState,
      id: generateVersionId(),
    };

    const db = await openDB('versions');
    const tx = db.transaction('versions', 'readwrite');
    await tx.objectStore('versions').add(version);

    // Cleanup old versions
    await this.cleanupOldVersions();
  }

  private async cleanupOldVersions() {
    const db = await openDB('versions');
    const versions = await db.getAll('versions');

    if (versions.length > this.maxVersions) {
      const toDelete = versions
        .sort((a, b) => a.timestamp - b.timestamp)
        .slice(0, versions.length - this.maxVersions);

      const tx = db.transaction('versions', 'readwrite');
      for (const version of toDelete) {
        await tx.objectStore('versions').delete(version.id);
      }
    }
  }
}
```

Use Cases

- Document editors (Google Docs-style auto-save)
- Long-form content creation
- Dashboard builder with complex state
- Applications with unreliable manual saves
- Compliance requirements (regular backups)

Advantages

- Predictable save behavior (users know when saves occur)
- Simple implementation (just setInterval)
- Works even if user forgets to save
- Can implement version history easily

Disadvantages

- May lose up to interval duration of work
- Wastes resources saving unchanged data
- Fixed interval may not match user activity patterns
- Can interfere with user actions if not careful

Best Practices

- Use 30-60 second intervals for documents
- Only save if data has changed (dirty flag)
- Show visual indicator of last auto-save time
- Pause auto-save during intensive operations
- Provide manual save option alongside auto-save

6.1.4.5 4. Manual Persistence

User explicitly triggers save operations via UI controls (buttons, keyboard shortcuts), giving users full control over when data is persisted.

Characteristics

- User initiates all save operations
- No automatic persistence
- Clear save/unsaved state indication
- Familiar desktop application model

Implementation Patterns

```
// Pattern 1: Basic manual save with dirty tracking
class ManualSaveManager {
  private isDirty = false;
  private currentData: any = null;
  private saveButton: HTMLButtonElement;

  constructor(saveButtonId: string) {
    this.saveButton = document.getElementById(saveButtonId) as HTMLButtonElement;
    this.saveButton.addEventListener('click', () => this.save());

    // Warn on navigation if unsaved changes
    window.addEventListener('beforeunload', (e) => {
      if (this.isDirty) {
        e.preventDefault();
        e.returnValue = 'You have unsaved changes. Are you sure you want to leave?';
      }
    });
  }

  onChange(data: any) {
    this.currentData = data;
    this.markDirty();
  }

  private markDirty() {
    this.isDirty = true;
```

```
    this.saveButton.disabled = false;
    this.saveButton.textContent = 'Save *';
    document.title = '* ' + document.title.replace(/^\* /, '');
  }

  private markClean() {
    this.isDirty = false;
    this.saveButton.disabled = true;
    this.saveButton.textContent = 'Saved';
    document.title = document.title.replace(/^\* /, '');
  }

  async save() {
    if (!this.isDirty) return;

    try {
      this.saveButton.textContent = 'Saving...';
      this.saveButton.disabled = true;

      await persistToIndexedDB(this.currentData);

      this.markClean();
      showNotification('Changes saved successfully');
    } catch (error) {
      this.saveButton.disabled = false;
      this.saveButton.textContent = 'Save (failed)';
      showErrorNotification('Failed to save changes');
      throw error;
    }
  }

  hasUnsavedChanges(): boolean {
    return this.isDirty;
  }
}

// Pattern 2: Keyboard shortcut support (Ctrl+S)
class KeyboardSaveManager extends ManualSaveManager {
  constructor(saveButtonId: string) {
    super(saveButtonId);

    document.addEventListener('keydown', (e) => {
      if ((e.ctrlKey || e.metaKey) && e.key === 's') {
        e.preventDefault();
        this.save();
      }
    });
  }
}

// Pattern 3: Command palette integration
```

```typescript
interface SaveCommand {
  id: string;
  name: string;
  shortcut: string;
  execute: () => Promise<void>;
}

class CommandBasedSave {
  private commands: Map<string, SaveCommand> = new Map();

  registerSaveCommand(command: SaveCommand) {
    this.commands.set(command.id, command);
  }

  async executeSave(commandId: string) {
    const command = this.commands.get(commandId);
    if (command) {
      await command.execute();
    }
  }
}

// Usage
const saveManager = new CommandBasedSave();

saveManager.registerSaveCommand({
  id: 'save-dashboard',
  name: 'Save Dashboard',
  shortcut: 'Ctrl+S',
  execute: async () => {
    await saveDashboard(getCurrentDashboard());
  }
});

saveManager.registerSaveCommand({
  id: 'save-as',
  name: 'Save Dashboard As...',
  shortcut: 'Ctrl+Shift+S',
  execute: async () => {
    const name = await promptForDashboardName();
    await saveDashboardAs(getCurrentDashboard(), name);
  }
});
```

Use Cases

- Power-user tools (IDEs, design software)
- Large documents where auto-save is expensive
- Applications with complex save logic
- When users need explicit control
- Desktop-like applications

Advantages

- User has full control over persistence
- No unexpected saves during editing
- Minimal performance overhead (save only when needed)
- Clear mental model (familiar to desktop users)

Disadvantages

- Users must remember to save (risk of data loss)
- Requires UI elements (save button, indicators)
- Poor UX for web applications (users expect auto-save)
- Need to handle navigation/close events

Best Practices

- Show clear unsaved changes indicator (* in title)
- Support Ctrl+S keyboard shortcut
- Warn on navigation if unsaved changes exist
- Disable save button when no changes
- Provide "Save As" for creating copies

6.1.4.6 5. Optimistic Persistence

Update UI immediately while persisting asynchronously in the background, providing the best user experience with eventual consistency guarantees.

Characteristics

- UI updates instantly (no waiting for persistence)
- Persistence happens asynchronously
- Requires rollback mechanism on failure
- Eventual consistency model

Implementation Patterns

```
// Pattern 1: Optimistic update with rollback
class OptimisticPersistence<T> {
  private previousState: T | null = null;

  async update(
    newState: T,
    updateUI: (state: T) => void,
    persist: (state: T) => Promise<void>
  ) {
    // Save current state for potential rollback
    this.previousState = getCurrentState<T>();

    // Update UI immediately
    updateUI(newState);

    try {
      // Persist asynchronously
      await persist(newState);
    } catch (error) {
```

```
      // Rollback on failure
      if (this.previousState) {
        updateUI(this.previousState);
        showErrorNotification('Failed to save changes. Reverted to previous state.');
      }
      throw error;
    }
  }
}

// Usage
const optimistic = new OptimisticPersistence<Dashboard>();

async function updateDashboardTitle(newTitle: string) {
  const dashboard = getCurrentDashboard();
  dashboard.title = newTitle;

  await optimistic.update(
    dashboard,
    (state) => renderDashboard(state),
    (state) => saveDashboardToServer(state)
  );
}

// Pattern 2: Queue-based optimistic updates
class OptimisticQueue<T> {
  private queue: Array<{
    id: string;
    operation: () => Promise<void>;
    rollback: () => void;
  }> = [];
  private processing = false;

  async enqueue(
    id: string,
    optimisticUpdate: () => void,
    persistOperation: () => Promise<void>,
    rollbackOperation: () => void
  ) {
    // Apply optimistic update immediately
    optimisticUpdate();

    // Queue persistence operation
    this.queue.push({
      id,
      operation: persistOperation,
      rollback: rollbackOperation,
    });

    // Process queue
    this.processQueue();
```

```
  }

  private async processQueue() {
    if (this.processing || this.queue.length === 0) return;

    this.processing = true;

    while (this.queue.length > 0) {
      const item = this.queue[0];

      try {
        await item.operation();
        this.queue.shift(); // Remove on success
      } catch (error) {
        // Rollback and remove from queue
        item.rollback();
        this.queue.shift();
        showErrorNotification(`Operation ${item.id} failed and was rolled back`);
      }
    }

    this.processing = false;
  }

  clear() {
    // Rollback all pending operations
    this.queue.forEach(item => item.rollback());
    this.queue = [];
  }
}

// Pattern 3: Collaborative optimistic updates with conflict resolution
class CollaborativeOptimistic {
  private localVersion = 0;
  private serverVersion = 0;

  async update(change: any) {
    const localId = ++this.localVersion;

    // Apply change locally
    applyChangeLocally(change, localId);

    try {
      // Send to server
      const result = await sendChangeToServer(change, localId);

      if (result.conflict) {
        // Server detected conflict, resolve it
        await this.resolveConflict(result.serverState, change);
      } else {
        // Success, update server version
```

```
        this.serverVersion = result.version;
      }
    } catch (error) {
      // Network error, queue for retry
      queueForRetry(change, localId);
    }
  }

  private async resolveConflict(serverState: any, localChange: any) {
    // Implement conflict resolution strategy
    const resolved = mergeChanges(serverState, localChange);
    applyChangeLocally(resolved, ++this.localVersion);
    await sendChangeToServer(resolved, this.localVersion);
  }
}
```

Use Cases

- Collaborative editing (Google Docs, Figma)
- Real-time dashboards with live updates
- Mobile applications with unreliable networks
- Social media interactions (likes, comments)
- Any UI requiring instant feedback

Advantages

- Best user experience (instant feedback)
- Works well with unreliable networks
- Enables offline-first applications
- Reduces perceived latency

Disadvantages

- Complex implementation (rollback logic)
- Requires conflict resolution for collaborative features
- May confuse users if rollbacks are frequent
- Difficult to debug

Best Practices

- Always save previous state for rollback
- Show subtle indicators for pending operations
- Implement retry logic for network failures
- Use operational transformation for conflicts
- Test rollback scenarios thoroughly

6.1.4.7 6. Lazy Persistence

Defer persistence until absolutely necessary (e.g., when data is about to be evicted from memory or user navigates away), minimizing storage operations.

Implementation

```
class LazyPersistence {
  private cache = new Map<string, any>();
```

```
  private dirtyKeys = new Set<string>();

  set(key: string, value: any) {
    this.cache.set(key, value);
    this.dirtyKeys.add(key);
  }

  get(key: string): any {
    return this.cache.get(key);
  }

  async flush() {
    const promises = Array.from(this.dirtyKeys).map(async (key) => {
      const value = this.cache.get(key);
      await persistToIndexedDB(key, value);
    });

    await Promise.all(promises);
    this.dirtyKeys.clear();
  }
}

// Flush on page unload
window.addEventListener('beforeunload', () => {
  lazyPersistence.flush();
});
```

Use Cases

- Non-critical cache data
- Temporary UI state
- Prefetched data
- Analytics events

6.1.4.8 7. Transactional Persistence

Group multiple related changes into atomic transactions that either all succeed or all fail, ensuring data consistency.

Implementation

```
class TransactionalPersistence {
  async executeTransaction(operations: Array<() => Promise<void>>) {
    const db = await openDB('app');
    const tx = db.transaction(['dashboards', 'settings'], 'readwrite');

    try {
      // Execute all operations within transaction
      for (const operation of operations) {
        await operation();
      }

      // Commit transaction
      await tx.done;
    } catch (error) {
```

```
      // Transaction automatically rolls back on error
      throw new Error('Transaction failed: ' + error.message);
    }
  }
}

// Usage
await transactional.executeTransaction([
  () => saveDashboard(dashboard),
  () => updateSettings(settings),
  () => logAuditEvent(event),
]);
```

Use Cases

- Multi-step data updates
- Ensuring referential integrity
- Financial transactions
- Critical state changes

6.1.5 Strategy Selection Guide

Choose persistence strategy based on these criteria:

Data Criticality

- **Critical** (settings, user data): Immediate or Transactional
- **Important** (dashboards, documents): Debounced or Periodic
- **Nice-to-have** (UI state, cache): Lazy or Optimistic

Update Frequency

- **High** (>10/sec): Debounced or Lazy
- **Medium** (1-10/sec): Debounced or Periodic
- **Low** (<1/sec): Immediate or Manual

User Expectations

- **Desktop-like**: Manual with Ctrl+S
- **Web-like**: Debounced or Periodic
- **Real-time**: Optimistic with sync

Network Dependency

- **Offline-capable**: Optimistic with queue
- **Online-only**: Immediate to server
- **Hybrid**: Optimistic local + eventual server sync

6.1.6 Storage Quota Management

Browser storage quotas represent one of the most significant constraints for client-side data persistence, requiring sophisticated management strategies to prevent data loss, maintain application functionality, and provide transparent user experiences. Unlike server-side databases with virtually unlimited storage, browser storage APIs impose strict limits that vary by browser, operating system, available disk space, and user settings.

These quotas are designed to prevent malicious websites from consuming excessive disk space, but they create challenges for legitimate applications—particularly data-intensive BI dashboards that cache large query results, store extensive dashboard configurations, and maintain offline-capable datasets.

The framework must implement proactive quota monitoring to detect approaching limits before they cause failures, intelligent cleanup policies to automatically reclaim space from stale or low-priority data, user-facing notifications that explain storage usage and provide actionable remediation options, and graceful degradation strategies that maintain core functionality even when storage is exhausted.

Effective quota management requires understanding browser-specific behaviors (Safari's aggressive eviction policies, Chrome's generous quotas, Firefox's user prompts), implementing prioritization schemes that preserve critical data while sacrificing cache, and providing escape hatches like cloud sync or data export when local storage proves insufficient.

6.1.6.1 Quota Limits by Browser

Different browsers implement vastly different storage quota policies, requiring applications to adapt their persistence strategies accordingly:

Desktop Browsers

Browser	Quota Calculation	Typical Limit	Eviction Policy	User Prompts
Chrome/Edge	~60% of available disk space	2-10GB+ (varies by disk)	Best-effort (rarely evicts)	No prompt for quota increase
Firefox	~50% of available disk space	2GB default, up to 8GB	Persistent (won't evict without permission)	Prompts at 50MB, 100MB thresholds
Safari	Fixed limit	~1GB	Aggressive (evicts after 7 days of inactivity)	Prompts for quota increase
Opera	Similar to Chrome	2-10GB+	Best-effort	No prompt

Mobile Browsers

Browser	Quota Calculation	Typical Limit	Eviction Policy	Considerations
Chrome Mobile	~60% of available space	50-500MB (device-dependent)	Best-effort	Limited by device storage
Safari iOS	Fixed limit	~50-100MB	Very aggressive (evicts quickly)	Extremely limited
Firefox Mobile	~50% of available space	100-500MB	Persistent	Better than Safari
Samsung Internet	Similar to Chrome	50-500MB	Best-effort	Android storage limits

Storage API Quota Estimation

```
interface StorageEstimate {
  quota?: number;      // Total available storage in bytes
  usage?: number;      // Currently used storage in bytes
  usageDetails?: {     // Breakdown by storage type
    indexedDB?: number;
    caches?: number;
    serviceWorkerRegistrations?: number;
```

```
  };
}

// Check current quota status
const estimate: StorageEstimate = await navigator.storage.estimate();
console.log(`Using ${estimate.usage} of ${estimate.quota} bytes`);
console.log(`Percentage used: ${(estimate.usage! / estimate.quota!) * 100}%`);
```

6.1.6.2 Quota Management Strategies

The framework implements a multi-layered approach to quota management, combining proactive monitoring, automatic cleanup, user notifications, and graceful degradation.

1. Proactive Monitoring

Continuously track storage usage and predict when quota limits will be reached, enabling preemptive action before failures occur.

```
class QuotaMonitor {
  private checkInterval = 60000; // Check every minute
  private thresholds = {
    warning: 0.75,    // 75% - show warning
    critical: 0.90,   // 90% - trigger cleanup
    emergency: 0.95,  // 95% - aggressive cleanup
  };
  private listeners: Array<(status: QuotaStatus) => void> = [];

  async start() {
    // Initial check
    await this.checkQuota();

    // Periodic monitoring
    setInterval(() => this.checkQuota(), this.checkInterval);

    // Check before large writes
    this.interceptStorageOperations();
  }

  private async checkQuota(): Promise<QuotaStatus> {
    const estimate = await navigator.storage.estimate();
    const percentUsed = (estimate.usage! / estimate.quota!) * 100;

    const status: QuotaStatus = {
      usage: estimate.usage!,
      quota: estimate.quota!,
      percentUsed,
      available: estimate.quota! - estimate.usage!,
      level: this.determineLevel(percentUsed),
      breakdown: estimate.usageDetails,
    };

    // Notify listeners
    this.notifyListeners(status);
```

```
  // Take action based on level
  await this.handleQuotaLevel(status);

  return status;
}

private determineLevel(percentUsed: number): QuotaLevel {
  if (percentUsed >= this.thresholds.emergency) return 'emergency';
  if (percentUsed >= this.thresholds.critical) return 'critical';
  if (percentUsed >= this.thresholds.warning) return 'warning';
  return 'normal';
}

private async handleQuotaLevel(status: QuotaStatus) {
  switch (status.level) {
    case 'warning':
      showNotification('Storage space running low', 'warning');
      break;

    case 'critical':
      showNotification('Storage space critically low. Cleaning up...', 'error');
      await this.performCleanup('moderate');
      break;

    case 'emergency':
      showNotification('Storage full! Performing emergency cleanup.', 'error');
      await this.performCleanup('aggressive');
      break;
  }
}

private interceptStorageOperations() {
  // Wrap IndexedDB operations to check quota before writes
  const originalPut = IDBObjectStore.prototype.put;
  IDBObjectStore.prototype.put = async function(value: any, key?: any) {
    const estimate = await navigator.storage.estimate();
    const percentUsed = (estimate.usage! / estimate.quota!) * 100;

    if (percentUsed > 95) {
      throw new Error('Storage quota exceeded. Cannot save data.');
    }

    return originalPut.call(this, value, key);
  };
}

subscribe(listener: (status: QuotaStatus) => void) {
  this.listeners.push(listener);
}
```

```
  private notifyListeners(status: QuotaStatus) {
    this.listeners.forEach(listener => listener(status));
  }
}

interface QuotaStatus {
  usage: number;
  quota: number;
  percentUsed: number;
  available: number;
  level: QuotaLevel;
  breakdown?: {
    indexedDB?: number;
    caches?: number;
    serviceWorkerRegistrations?: number;
  };
}

type QuotaLevel = 'normal' | 'warning' | 'critical' | 'emergency';
```

2. Automatic Cleanup

Implement intelligent cleanup policies that automatically reclaim storage space from low-priority or stale data.

```
class StorageCleanupManager {
  private cleanupPolicies: CleanupPolicy[] = [
    // Policy 1: TTL-based expiration
    {
      name: 'expired-cache',
      priority: 1,
      execute: async () => {
        const db = await openDB('cache');
        const now = Date.now();
        const expired = await db.getAllFromIndex(
          'cache',
          'expiresAt',
          IDBKeyRange.upperBound(now)
        );

        for (const item of expired) {
          await db.delete('cache', item.id);
        }

        return { deletedCount: expired.length, bytesFreed: this.calculateSize(expired) };
      },
    },

    // Policy 2: LRU eviction for query results
    {
      name: 'lru-queries',
      priority: 2,
      execute: async () => {
```

```
      const db = await openDB('queries');
      const queries = await db.getAll('queries');

      // Sort by last access time
      queries.sort((a, b) => a.lastAccessedAt - b.lastAccessedAt);

      // Remove oldest 25%
      const toDelete = queries.slice(0, Math.floor(queries.length * 0.25));

      for (const query of toDelete) {
        await db.delete('queries', query.id);
      }

      return { deletedCount: toDelete.length, bytesFreed: this.calculateSize(toDelete) };
    },
  },

  // Policy 3: Compress large objects
  {
    name: 'compress-large-objects',
    priority: 3,
    execute: async () => {
      const db = await openDB('dashboards');
      const dashboards = await db.getAll('dashboards');
      let bytesFreed = 0;

      for (const dashboard of dashboards) {
        const serialized = JSON.stringify(dashboard);
        if (serialized.length > 100000) { // > 100KB
          const compressed = await this.compress(serialized);
          const savings = serialized.length - compressed.length;

          if (savings > 0) {
            await db.put('dashboards', {
              ...dashboard,
              _compressed: true,
              data: compressed,
            });
            bytesFreed += savings;
          }
        }
      }

      return { deletedCount: 0, bytesFreed };
    },
  },

  // Policy 4: Remove old versions
  {
    name: 'old-versions',
    priority: 4,
```

```
    execute: async () => {
      const db = await openDB('versions');
      const versions = await db.getAll('versions');

      // Keep only last 5 versions per dashboard
      const grouped = this.groupBy(versions, 'dashboardId');
      let deletedCount = 0;
      let bytesFreed = 0;

      for (const [dashboardId, dashboardVersions] of Object.entries(grouped)) {
        const sorted = dashboardVersions.sort((a, b) => b.timestamp - a.timestamp);
        const toDelete = sorted.slice(5); // Keep 5 most recent

        for (const version of toDelete) {
          await db.delete('versions', version.id);
          deletedCount++;
          bytesFreed += this.calculateSize([version]);
        }
      }

      return { deletedCount, bytesFreed };
    },
  },
];

async performCleanup(intensity: 'light' | 'moderate' | 'aggressive'): Promise<CleanupResult> {
  const policiesToRun = this.selectPolicies(intensity);
  const results: CleanupResult[] = [];

  for (const policy of policiesToRun) {
    try {
      const result = await policy.execute();
      results.push({ policy: policy.name, ...result });
      console.log(`Cleanup policy '${policy.name}': deleted ${result.deletedCount} items, freed ${result.by
    } catch (error) {
      console.error(`Cleanup policy '${policy.name}' failed:`, error);
    }
  }

  const totalFreed = results.reduce((sum, r) => sum + r.bytesFreed, 0);
  const totalDeleted = results.reduce((sum, r) => sum + r.deletedCount, 0);

  showNotification(`Cleanup complete: freed ${this.formatBytes(totalFreed)}, removed ${totalDeleted} items`

  return { policy: 'combined', deletedCount: totalDeleted, bytesFreed: totalFreed };
}

private selectPolicies(intensity: 'light' | 'moderate' | 'aggressive'): CleanupPolicy[] {
  switch (intensity) {
    case 'light':
      return this.cleanupPolicies.filter(p => p.priority <= 2);
```

```
      case 'moderate':
        return this.cleanupPolicies.filter(p => p.priority <= 3);
      case 'aggressive':
        return this.cleanupPolicies;
    }
  }

  private async compress(data: string): Promise<ArrayBuffer> {
    const blob = new Blob([data]);
    const stream = blob.stream().pipeThrough(new CompressionStream('gzip'));
    const compressed = await new Response(stream).arrayBuffer();
    return compressed;
  }

  private calculateSize(items: any[]): number {
    return items.reduce((sum, item) => {
      return sum + JSON.stringify(item).length;
    }, 0);
  }

  private formatBytes(bytes: number): string {
    if (bytes < 1024) return bytes + ' B';
    if (bytes < 1024 * 1024) return (bytes / 1024).toFixed(2) + ' KB';
    return (bytes / (1024 * 1024)).toFixed(2) + ' MB';
  }

  private groupBy<T>(array: T[], key: keyof T): Record<string, T[]> {
    return array.reduce((groups, item) => {
      const groupKey = String(item[key]);
      if (!groups[groupKey]) groups[groupKey] = [];
      groups[groupKey].push(item);
      return groups;
    }, {} as Record<string, T[]>);
  }
}

interface CleanupPolicy {
  name: string;
  priority: number;
  execute: () => Promise<{ deletedCount: number; bytesFreed: number }>;
}

interface CleanupResult {
  policy: string;
  deletedCount: number;
  bytesFreed: number;
}
```

3. User Notifications

Provide transparent, actionable notifications that help users understand and manage storage usage.

```
class StorageNotificationManager {
  showQuotaWarning(status: QuotaStatus) {
    const message = `
      Storage space is running low (${status.percentUsed.toFixed(1)}% used).

      Current usage: ${this.formatBytes(status.usage)}
      Available: ${this.formatBytes(status.available)}

      Consider:
      • Clearing old query results
      • Removing unused dashboards
      • Enabling cloud sync
    `;

    showNotificationWithActions(message, [
      {
        label: 'View Storage Usage',
        action: () => this.showStorageBreakdown(status),
      },
      {
        label: 'Clean Up Now',
        action: () => this.performCleanup(),
      },
      {
        label: 'Enable Cloud Sync',
        action: () => this.enableCloudSync(),
      },
    ]);
  }

  showStorageBreakdown(status: QuotaStatus) {
    const breakdown = status.breakdown || {};
    const total = status.usage;

    const chart = [
      { label: 'IndexedDB', bytes: breakdown.indexedDB || 0, percent: ((breakdown.indexedDB || 0) / total) *
      { label: 'Cache API', bytes: breakdown.caches || 0, percent: ((breakdown.caches || 0) / total) * 100 },
      { label: 'Service Workers', bytes: breakdown.serviceWorkerRegistrations || 0, percent: ((breakdown.serv
    ];

    // Show modal with breakdown
    showModal({
      title: 'Storage Usage Breakdown',
      content: this.renderBreakdownChart(chart),
      actions: [
        { label: 'Close', action: 'close' },
        { label: 'Clear Cache', action: () => this.clearCache() },
      ],
    });
  }
```

```
  private formatBytes(bytes: number): string {
    if (bytes < 1024) return bytes + ' B';
    if (bytes < 1024 * 1024) return (bytes / 1024).toFixed(2) + ' KB';
    return (bytes / (1024 * 1024)).toFixed(2) + ' MB';
  }
}
```

4. Graceful Degradation

Maintain core functionality even when storage quota is exceeded by implementing fallback strategies.

```
class GracefulDegradationManager {
  private quotaExceeded = false;
  private fallbackMode: 'memory-only' | 'server-only' | 'disabled' = 'memory-only';

  async handleQuotaExceeded() {
    this.quotaExceeded = true;

    // Disable non-essential caching
    this.disableQueryCache();

    // Switch to memory-only mode
    this.enableMemoryOnlyMode();

    // Notify user
    showNotification(
      'Storage quota exceeded. Running in limited mode. Some features may be unavailable.',
      'warning'
    );

    // Attempt cleanup
    await this.attemptRecovery();
  }

  private disableQueryCache() {
    // Disable caching of query results
    window.QUERY_CACHE_ENABLED = false;

    // Clear existing cache to free space
    this.clearQueryCache();
  }

  private enableMemoryOnlyMode() {
    // Store data in memory instead of IndexedDB
    this.fallbackMode = 'memory-only';

    // Warn about data loss on refresh
    window.addEventListener('beforeunload', (e) => {
      if (this.quotaExceeded) {
        e.preventDefault();
        e.returnValue = 'Data will be lost on refresh due to storage limitations.';
      }
```

```
    });
  }

  private async attemptRecovery() {
    // Try aggressive cleanup
    const cleanupManager = new StorageCleanupManager();
    await cleanupManager.performCleanup('aggressive');

    // Check if we recovered enough space
    const estimate = await navigator.storage.estimate();
    const percentUsed = (estimate.usage! / estimate.quota!) * 100;

    if (percentUsed < 90) {
      this.quotaExceeded = false;
      this.fallbackMode = 'memory-only';
      showNotification('Storage space recovered. Normal operation resumed.', 'success');
    }
  }

  async saveWithFallback(key: string, value: any) {
    if (this.quotaExceeded && this.fallbackMode === 'memory-only') {
      // Store in memory only
      memoryStore.set(key, value);
      return;
    }

    try {
      // Try IndexedDB
      await saveToIndexedDB(key, value);
    } catch (error) {
      if (error.name === 'QuotaExceededError') {
        await this.handleQuotaExceeded();
        // Fallback to memory
        memoryStore.set(key, value);
      } else {
        throw error;
      }
    }
  }
}
```

6.1.6.3 Browser-Specific Considerations

Safari

- Extremely aggressive eviction (7 days of inactivity)
- Request persistent storage: `await navigator.storage.persist()`
- Warn users about Safari's limitations
- Recommend cloud sync for Safari users

Firefox

- Prompts users at 50MB and 100MB thresholds

- Persistent storage prevents eviction
- Better quota management than Safari

Chrome/Edge

- Generous quotas (60% of disk space)
- Rarely evicts data
- Best-effort storage by default
- Request persistent storage for critical apps

Mobile Browsers

- Much stricter limits (50-500MB)
- More aggressive eviction
- Prioritize critical data only
- Implement aggressive cleanup policies

6.1.7 Security and Privacy

Data persistence in browser storage introduces significant security and privacy challenges that must be carefully addressed to protect sensitive user data, comply with regulations like GDPR and CCPA, and prevent unauthorized access or data leakage.

Unlike server-side databases protected by firewalls, authentication layers, and network isolation, browser storage is inherently more vulnerable—data persists on user devices that may be shared, stolen, or compromised, storage APIs are accessible to any JavaScript code running in the same origin, and browser developer tools provide direct access to inspect and modify stored data.

The framework must implement defense-in-depth security strategies including encryption at rest for sensitive data, strict data isolation between users and workspaces, comprehensive privacy controls that respect user preferences and regulatory requirements, and secure transmission protocols for synchronizing data with remote servers. Security considerations extend beyond technical controls to include user education (warning about shared devices), audit logging (tracking who accessed what data when), and incident response procedures (detecting and responding to potential breaches).

Privacy compliance requires implementing data minimization (only store necessary data), retention policies (automatically delete old data), user rights (export, deletion, portability), and transparency (clear privacy policies explaining what data is stored and why).

6.1.7.1 1. Encryption at Rest

Protect sensitive data stored in IndexedDB by encrypting it before persistence, ensuring that even if an attacker gains access to the browser storage, they cannot read the data without the encryption key.

Encryption Architecture

```
class StorageEncryptionService {
  private algorithm = 'AES-GCM';
  private keyLength = 256;
  private ivLength = 12; // 96 bits for GCM

  /**


   * Generate encryption key from user password using PBKDF2
   */
  async deriveKeyFromPassword(password: string, salt: Uint8Array): Promise<CryptoKey> {
    const encoder = new TextEncoder();
```

```
    const passwordBuffer = encoder.encode(password);

    // Import password as key material
    const keyMaterial = await crypto.subtle.importKey(
      'raw',
      passwordBuffer,
      'PBKDF2',
      false,
      ['deriveKey']
    );

    // Derive AES key using PBKDF2
    const key = await crypto.subtle.deriveKey(
      {
        name: 'PBKDF2',
        salt: salt,
        iterations: 100000, // OWASP recommendation
        hash: 'SHA-256',
      },
      keyMaterial,
      {
        name: this.algorithm,
        length: this.keyLength,
      },
      false, // Not extractable
      ['encrypt', 'decrypt']
    );

    return key;
}\n\n  /**\n   * Generate random encryption key (for session-based encryption)\n   */\n  async generateKey():

Invalid key or corrupted data.');\n    }\n  }\n\n  /**\n   * Store encrypted data in IndexedDB\n   */\n  asyn

new Uint8Array(JSON.parse(stored)) : null;\n  }\n\n  private async storeSalt(salt: Uint8Array) {\n    localSt

Data Isolation\n\nEnsure strict separation of data between users, workspaces, and tenants to prevent unauthor

Privacy Compliance\n\nImplement GDPR, CCPA, and other privacy regulations' requirements for user data managem

Secure Transmission\n\nProtect data during synchronization with remote servers using encryption, authenticati

HTTPS required.');\n    }\n    \n    // Get authentication token\n    const token = await this.getAuthToken()

Never Store Sensitive Data in Plain Text**\n- Always encrypt API keys, passwords, tokens\n- Use Web Crypto AP

Sanitize User Input**\n```typescript\nfunction sanitizeInput(input: string): string {\n  return input\n    .r

Implement Rate Limiting**\n```typescript\nclass RateLimiter {\n  private attempts = new Map<string, number[]>

Clear Sensitive Data on Logout**\n```typescript\nasync function logout() {\n  // Clear all storage\n  await c
```

6.1.8 Performance Considerations

Persistence operations represent a critical performance bottleneck in client-side applications, with the potential to significantly impact user experience through UI freezes, slow page loads, and unresponsive interactions. Unlike server-side databases optimized for concurrent access and high throughput, browser storage APIs (localStorage, IndexedDB, Cache API) operate on the main thread or with limited concurrency, making poorly optimized persistence code a common source of performance problems.

The framework must implement comprehensive performance optimizations across read operations (minimizing database queries, caching frequently accessed data, using indexes effectively), write operations (batching transactions, debouncing updates, compressing payloads), memory management (avoiding memory leaks, streaming large datasets, triggering garbage collection), and network synchronization (delta updates, compression, request batching).

Performance monitoring and profiling are essential—measuring persistence latency, tracking memory usage, identifying slow queries, and optimizing hot paths based on real-world usage patterns rather than assumptions.

6.1.8.1 1. Read Optimization

Optimize data retrieval to minimize latency and maximize throughput for frequently accessed data.

Indexing Strategy

```
// Create indexes for frequently queried fields
const db = await openDB('dashboards', 1, {
  upgrade(db) {
    const dashboardStore = db.createObjectStore('dashboards', { keyPath: 'id' });
    \n    // Index by user ID for fast user-specific queries
    dashboardStore.createIndex('userId', 'userId', { unique: false });
    \n    // Index by creation date for time-based queries
    dashboardStore.createIndex('createdAt', 'createdAt', { unique: false });
    \n    // Compound index for user + date queries
    dashboardStore.createIndex('userIdCreatedAt', ['userId', 'createdAt'], { unique: false });
    \n    // Index by tags for filtering
    dashboardStore.createIndex('tags', 'tags', { unique: false, multiEntry: true });
  },
});\n\n// Use indexes for fast queries
async function getDashboardsByUser(userId: string): Promise<Dashboard[]> {
  const db = await openDB('dashboards');
  return await db.getAllFromIndex('dashboards', 'userId', userId);
}\n\n// Range queries using indexes
async function getRecentDashboards(userId: string, days: number = 7): Promise<Dashboard[]> {
  const db = await openDB('dashboards');
  const cutoff = Date.now() - (days * 24 * 60 * 60 * 1000);
\n  return await db.getAllFromIndex(\n    'dashboards',\n    'userIdCreatedAt',\n    IDBKeyRange.bound([userI

Write Optimization\n\nOptimize data persistence to minimize write latency and avoid blocking the UI.\n\n**Tra

Memory Management\n\nPrevent memory leaks and optimize memory usage for large datasets.\n\n**Streaming Large

Triggering cleanup...');\n        await this.cleanup();\n      }\n    }\n  }\n  \n  private async cleanup() {

Network Optimization\n\nOptimize data synchronization with remote servers to minimize bandwidth and latency.\
```

6.2 Storage Strategy & Configuration

6.3 User Configurations

6.3.1 Purpose

Persistent storage and management of user preferences, application settings, and workspace configurations for the BI Dashboard Framework. User Configurations provide a centralized, type-safe mechanism for storing and retrieving user-specific data that persists across sessions, enabling personalized experiences tailored to individual workflows, preferences, and organizational requirements.

This system manages everything from simple UI preferences (theme selection, panel layouts, default chart types) to complex workspace configurations (saved queries, custom filters, dashboard templates, data source credentials) to advanced behavioral settings (auto-refresh intervals, notification preferences, keyboard shortcuts, accessibility options).

The configuration system is designed with multi-tenancy in mind, supporting user-level, workspace-level, and organization-level settings with proper inheritance and override semantics, ensuring that administrators can enforce organizational policies while users retain control over personal preferences. All configurations are validated against schemas to prevent invalid states, versioned to support migration across application updates, and encrypted when containing sensitive data like API keys or database credentials.

The system provides reactive updates—when a setting changes, all dependent UI components automatically re-render with the new values—eliminating the need for manual refresh or application restart. Export and import capabilities enable users to backup configurations, share workspace setups with colleagues, or migrate settings between environments, while audit logging tracks configuration changes for compliance and debugging purposes.

6.3.2 Features

- **Hierarchical Settings Organization**: Nested configuration structure organized by domain (appearance, behavior, data, security)
- **Type-Safe Schema Validation**: Runtime validation using Zod schemas with TypeScript type inference
- **Multi-Level Inheritance**: User settings override workspace defaults, which override organization policies
- **Reactive Updates**: Automatic UI re-rendering when settings change via Zustand subscriptions
- **Persistent Storage**: IndexedDB for client-side persistence with optional cloud sync
- **Import/Export**: JSON-based configuration backup and sharing
- **Encryption**: Sensitive settings encrypted at rest using Web Crypto API
- **Versioning**: Schema versioning with automatic migration for backward compatibility
- **Audit Logging**: Track configuration changes with timestamps and user attribution
- **Default Management**: Intelligent defaults with reset-to-default functionality

6.3.3 Configuration Categories

1. **Appearance Settings**
    - Theme selection (light, dark, high-contrast, custom)
    - Color scheme customization
    - Font family and size preferences
    - Panel layout and sidebar visibility
    - Dashboard grid density
    - Icon set selection
2. **Behavior Settings**
    - Auto-save interval for dashboards
    - Default chart types for data visualizations
    - Drag-and-drop sensitivity

- Confirmation dialogs (enable/disable)
- Undo/redo history depth
- Default data refresh intervals

3. **Data Settings**
   - Default data source connections
   - Query timeout limits
   - Cache expiration policies
   - Data sampling preferences
   - Export format defaults (CSV, Excel, JSON)
   - Date/time format and timezone
4. **Security Settings**
   - Session timeout duration
   - Two-factor authentication preferences
   - API key management
   - Data access permissions
   - Audit log retention
   - Encryption preferences
5. **Accessibility Settings**
   - Screen reader support
   - Keyboard navigation preferences
   - High-contrast mode
   - Animation reduction
   - Font scaling
   - Focus indicators
6. **Notification Settings**
   - Email notification preferences
   - In-app notification types
   - Alert thresholds for data anomalies
   - Scheduled report delivery
   - Collaboration notifications
   - System update alerts

6.3.4 Storage Architecture

The configuration system uses a multi-tiered storage approach combining in-memory state, browser storage, and optional cloud persistence:

In-Memory Layer (Zustand Store)

```
interface UserConfigStore {
  appearance: AppearanceConfig;
  behavior: BehaviorConfig;
  data: DataConfig;
  security: SecurityConfig;
  accessibility: AccessibilityConfig;
  notifications: NotificationConfig;

  // Actions
  updateConfig: <T extends keyof UserConfigStore>(
```

```
    category: T,
    updates: Partial<UserConfigStore[T]>
  ) => void;
  resetToDefaults: (category?: keyof UserConfigStore) => void;
  exportConfig: () => string;
  importConfig: (json: string) => void;
}
```

Persistence Layer (IndexedDB)

- Stores serialized configuration objects
- Supports offline-first operation
- Enables large configuration storage (>10MB)
- Provides transaction-based updates

Cloud Sync Layer (Optional)

- Syncs configurations across devices
- Enables team workspace sharing
- Provides version history and rollback
- Requires authentication and authorization

6.3.5 Implementation Example

```
import { create } from 'zustand';
import { persist } from 'zustand/middleware';
import { z } from 'zod';

// Schema definition
const AppearanceConfigSchema = z.object({
  theme: z.enum(['light', 'dark', 'high-contrast', 'custom']),
  colorScheme: z.string().optional(),
  fontSize: z.number().min(10).max(24).default(14),
  sidebarVisible: z.boolean().default(true),
  gridDensity: z.enum(['compact', 'normal', 'comfortable']).default('normal'),
});

type AppearanceConfig = z.infer<typeof AppearanceConfigSchema>;

// Store creation
const useConfigStore = create(
  persist<UserConfigStore>(
    (set, get) => ({
      appearance: AppearanceConfigSchema.parse({}),
      // ... other categories

      updateConfig: (category, updates) => {
        const current = get()[category];
        const merged = { ...current, ...updates };

        // Validate against schema
        const schema = getSchemaForCategory(category);
```

```
        const validated = schema.parse(merged);

        set({ [category]: validated });

        // Trigger persistence
        persistToIndexedDB(category, validated);
      },

      resetToDefaults: (category) => {
        if (category) {
          const defaults = getDefaultsForCategory(category);
          set({ [category]: defaults });
        } else {
          set(getAllDefaults());
        }
      },

      exportConfig: () => {
        const config = get();
        return JSON.stringify(config, null, 2);
      },

      importConfig: (json) => {
        const parsed = JSON.parse(json);
        // Validate all categories
        const validated = validateAllCategories(parsed);
        set(validated);
      },
    }),
    {
      name: 'user-config-storage',
      storage: createIndexedDBStorage(),
    }
  )
);
```

6.3.6 Migration Strategy

As the infinite canvas dashboard evolves, data schemas for cells, dashboards, and configurations will change. A migration strategy ensures existing user data remains compatible with new versions while enabling feature improvements. This section addresses key migration concerns specific to canvas-based dashboards with dynamic cell execution.

6.3.6.1 Key Migration Questions and Solutions

6.3.6.1.1 1. Canvas State Evolution **Question**: When you add new features to cells (e.g., new cell types, new properties), how should existing saved dashboards be handled?

Solution: Graceful degradation with automatic schema enrichment

```
interface Dashboard {
  version: number;
  cells: Cell[];
  metadata: DashboardMetadata;
```

```
}

interface Cell {
  id: string;
  type: string;
  position: { x: number; y: number; width: number; height: number };
  data: any;
  // New fields added over time
  permissions?: string[]; // Added in v2
  cachePolicy?: CachePolicy; // Added in v3
}

class DashboardLoader {
  private currentVersion = 3;

  async loadDashboard(dashboardId: string): Promise<Dashboard> {
    const raw = await this.storage.get(dashboardId);

    // No version = v1 (legacy)
    const version = raw.version || 1;

    if (version === this.currentVersion) {
      return raw;
    }

    // Auto-migrate with sensible defaults
    return this.migrateDashboard(raw, version);
  }

  private migrateDashboard(dashboard: any, fromVersion: number): Dashboard {
    let migrated = { ...dashboard };

    // v1 -> v2: Add permissions to cells
    if (fromVersion < 2) {
      migrated.cells = migrated.cells.map(cell => ({
        ...cell,
        permissions: ['execute', 'edit'] // Default permissions
      }));
      migrated.version = 2;
    }

    // v2 -> v3: Add cache policy
    if (fromVersion < 3) {
      migrated.cells = migrated.cells.map(cell => ({
        ...cell,
        cachePolicy: { enabled: true, ttl: 300000 } // 5 min default
      }));
      migrated.version = 3;
    }

    return migrated;
```

```
  }
}
```

Approach: Silent auto-migration with defaults for missing fields. No user intervention required.

6.3.6.1.2 2. Cell Data Schema Changes **Question**: If cell data structure changes (e.g., rename `cell.config` to `cell.settings`), how should old data be transformed?

Solution: Field mapping with backward compatibility layer

```
class CellMigrator {
  migrateCell(cell: any, fromVersion: number): Cell {
    let migrated = { ...cell };

    // v1 -> v2: Rename config to settings
    if (fromVersion < 2 && migrated.config) {
      migrated.settings = migrated.config;
      delete migrated.config;
    }

    // v2 -> v3: Restructure nested data
    if (fromVersion < 3 && migrated.settings?.display) {
      migrated.appearance = {
        ...migrated.settings.display,
        theme: migrated.settings.theme
      };
      delete migrated.settings.display;
      delete migrated.settings.theme;
    }

    return migrated;
  }

  // Backward compatibility: Support old field names in read operations
  getCellConfig(cell: Cell): any {
    // Try new field first, fall back to old
    return cell.settings || cell.config || {};
  }
}
```

Approach: Transform data on load, maintain backward-compatible getters during transition period.

6.3.6.1.3 3. Breaking Changes **Question**: When removing a cell type or feature, how should existing cells be handled?

Solution: Deprecation with fallback rendering

```
class CellRenderer {
  private deprecatedTypes = new Map<string, string>([
    ['legacy-chart', 'chart'], // Map to replacement
    ['old-table', 'table'],
  ]);

  renderCell(cell: Cell): HTMLElement {
```

```
    // Check if cell type is deprecated
    if (this.deprecatedTypes.has(cell.type)) {
      const replacement = this.deprecatedTypes.get(cell.type)!;

      // Show deprecation warning
      this.showDeprecationWarning(cell.id, cell.type, replacement);

      // Attempt to render with replacement type
      return this.renderWithType({ ...cell, type: replacement });
    }

    // Check if cell type no longer exists
    if (!this.cellRegistry.has(cell.type)) {
      return this.renderUnsupportedCell(cell);
    }

    return this.renderWithType(cell);
  }

  private renderUnsupportedCell(cell: Cell): HTMLElement {
    const container = document.createElement('div');
    container.className = 'cell-unsupported';
    container.innerHTML = `
      <div class="warning">
        <h3>Unsupported Cell Type: ${cell.type}</h3>
        <p>This cell type is no longer supported.</p>
        <button onclick="convertCell('${cell.id}')">Convert to Text</button>
        <button onclick="deleteCell('${cell.id}')">Delete Cell</button>
      </div>
      <details>
        <summary>View Raw Data</summary>
        <pre>${JSON.stringify(cell.data, null, 2)}</pre>
      </details>
    `;
    return container;
  }
}
```

Approach: Preserve data, show warnings, offer conversion options. Never silently delete user data.

6.3.6.1.4 4. Version Tracking

Question: Should dashboards track schema versions?

Solution: Yes, with metadata for debugging and migration tracking

```
interface DashboardMetadata {
  version: number;
  createdAt: number;
  updatedAt: number;
  createdWith: string; // App version that created it
  lastMigratedAt?: number;
  lastMigratedFrom?: number;
  migrationHistory?: MigrationRecord[];
```

```
}

interface MigrationRecord {
  from: number;
  to: number;
  timestamp: number;
  success: boolean;
  error?: string;
}

class DashboardVersionManager {
  async saveDashboard(dashboard: Dashboard) {
    const metadata: DashboardMetadata = {
      version: this.currentVersion,
      createdAt: dashboard.metadata.createdAt || Date.now(),
      updatedAt: Date.now(),
      createdWith: dashboard.metadata.createdWith || APP_VERSION,
      lastMigratedAt: dashboard.metadata.lastMigratedAt,
      lastMigratedFrom: dashboard.metadata.lastMigratedFrom,
      migrationHistory: dashboard.metadata.migrationHistory || []
    };

    await this.storage.set(dashboard.id, {
      ...dashboard,
      metadata
    });
  }

  recordMigration(dashboard: Dashboard, from: number, to: number, success: boolean, error?: string) {
    dashboard.metadata.migrationHistory = dashboard.metadata.migrationHistory || [];
    dashboard.metadata.migrationHistory.push({
      from,
      to,
      timestamp: Date.now(),
      success,
      error
    });

    if (success) {
      dashboard.metadata.lastMigratedAt = Date.now();
      dashboard.metadata.lastMigratedFrom = from;
    }
  }
}
```

Approach: Track versions and migration history for debugging and rollback capabilities.

6.3.6.1.5 5. User Experience **Question**: How should users be notified during migration?

Solution: Progressive disclosure with optional backup

```
class MigrationUI {
  async loadDashboardWithMigration(dashboardId: string): Promise<Dashboard> {
    const raw = await this.storage.get(dashboardId);
    const version = raw.version || 1;

    if (version === this.currentVersion) {
      return raw;
    }

    // Check if migration is significant
    const isSignificant = this.isSignificantMigration(version, this.currentVersion);

    if (isSignificant) {
      // Show migration dialog
      const userConsent = await this.showMigrationDialog(version, this.currentVersion);

      if (!userConsent) {
        throw new Error('Migration cancelled by user');
      }

      // Create backup
      await this.createBackup(dashboardId, raw);
    }

    // Show progress for long migrations
    const migrated = await this.migrateWithProgress(raw, version);

    // Show success notification
    this.showMigrationSuccess(version, this.currentVersion);

    return migrated;
  }

  private async showMigrationDialog(from: number, to: number): Promise<boolean> {
    return new Promise(resolve => {
      const dialog = document.createElement('div');
      dialog.className = 'migration-dialog';
      dialog.innerHTML = `
        <h2>Dashboard Update Required</h2>
        <p>This dashboard was created with an older version (v${from}).</p>
        <p>It will be upgraded to v${to} to support new features.</p>
        <p><strong>A backup will be created automatically.</strong></p>
        <div class="actions">
          <button id="migrate-yes">Update Dashboard</button>
          <button id="migrate-no">Cancel</button>
        </div>
      `;

      document.body.appendChild(dialog);

      dialog.querySelector('#migrate-yes')?.addEventListener('click', () => {
```

```
        dialog.remove();
        resolve(true);
      });

      dialog.querySelector('#migrate-no')?.addEventListener('click', () => {
        dialog.remove();
        resolve(false);
      });
    });
  }

  private async createBackup(dashboardId: string, dashboard: any) {
    const backupId = `${dashboardId}_backup_${Date.now()}`;
    await this.storage.set(backupId, {
      ...dashboard,
      _isBackup: true,
      _originalId: dashboardId
    });

    this.showNotification(`Backup created: ${backupId}`);
  }
}
```

Approach: Silent migration for minor changes, user consent + backup for major changes.

6.3.6.1.6 6. Scope of Migration

Question: What data needs migration?

Solution: Tiered migration strategy based on data type

```
interface MigrationScope {
  dashboards: boolean;      // Canvas state, cell positions
  cellData: boolean;         // Cell-specific data
  executionResults: boolean; // Cached execution outputs
  userPreferences: boolean;  // Settings and preferences
  queryCache: boolean;       // Cached query results
}

class ScopedMigrationManager {
  async migrateAll(scope: MigrationScope = { dashboards: true, cellData: true, executionResults: false, userP
    const migrations: Promise<void>[] = [];

    if (scope.dashboards) {
      migrations.push(this.migrateDashboards());
    }

    if (scope.cellData) {
      migrations.push(this.migrateCellData());
    }

    if (scope.userPreferences) {
      migrations.push(this.migrateUserPreferences());
    }
```

```
    // Execution results and query cache are ephemeral - just clear them
    if (scope.executionResults) {
      await this.clearExecutionResults();
    }

    if (scope.queryCache) {
      await this.clearQueryCache();
    }

    await Promise.all(migrations);
  }

  private async migrateDashboards() {
    const dashboards = await this.storage.getAllDashboards();

    for (const dashboard of dashboards) {
      const version = dashboard.version || 1;
      if (version < this.currentVersion) {
        const migrated = await this.dashboardMigrator.migrate(dashboard, version);
        await this.storage.saveDashboard(migrated);
      }
    }
  }

  private async clearExecutionResults() {
    // Execution results are tied to code versions - safer to regenerate
    await this.storage.clearNamespace('execution-results');
  }

  private async clearQueryCache() {
    // Query cache may have schema changes - safer to regenerate
    await this.storage.clearNamespace('query-cache');
  }
}
```

Approach:

- **Migrate**: Dashboards, cell data, user preferences (persistent, user-created)
- **Clear & regenerate**: Execution results, query cache (ephemeral, derived data)

6.3.6.2 Migration Best Practices

1. **Never delete user data** - Always preserve or offer conversion
2. **Provide sensible defaults** for new fields
3. **Create backups** before significant migrations
4. **Track migration history** for debugging
5. **Test with real data** from production
6. **Make migrations idempotent** (safe to run multiple times)
7. **Version everything** - dashboards, cells, configurations

6.4 Settings Management

6.4.1 Purpose

Comprehensive infrastructure for managing, persisting, and synchronizing application settings across the BI Dashboard Framework. Settings Management provides the technical foundation that powers the User Configurations system, implementing the low-level mechanisms for storing preferences, validating changes, propagating updates, and maintaining consistency across distributed components.

While User Configurations focus on what settings exist and their business semantics, Settings Management addresses how those settings are stored, retrieved, updated, and synchronized in a performant, reliable, and scalable manner.

This system handles the complex orchestration of multiple storage backends (in-memory state, localStorage, IndexedDB, remote servers), manages the lifecycle of settings from initialization through updates to persistence, implements optimistic updates with rollback capabilities for network failures, and provides reactive subscriptions that automatically notify components when relevant settings change.

The architecture supports advanced scenarios like settings inheritance hierarchies (organization → workspace → user), conflict resolution for concurrent updates in collaborative environments, partial updates to minimize network traffic, and efficient caching strategies to reduce database queries.

Settings Management also implements critical cross-cutting concerns including encryption for sensitive data, compression for large configuration objects, versioning for schema evolution, and audit logging for compliance tracking, ensuring that the settings infrastructure meets enterprise requirements for security, performance, and observability.

6.4.2 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

6.4.3 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

Approach	Pros	Cons	Best For
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

6.4.4 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

6.4.5 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

6.4.6 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

6.4.7 Recommended Architecture

The recommended architecture for Settings Management in the BI Dashboard Framework combines proven technologies and patterns to deliver a robust, performant, and maintainable configuration system.

This architecture leverages Zustand for reactive state management (lightweight, minimal boilerplate, excellent TypeScript support), Zod for runtime schema validation (type inference, clear error messages, composable schemas), hybrid persistence strategies (immediate persistence for critical settings, debounced persistence for frequently changing UI preferences), and IndexedDB for client-side storage (large capacity, indexed queries, transaction support).

The architecture is designed to scale from simple single-user scenarios to complex multi-tenant enterprise deployments, supporting features like settings inheritance (organization → workspace → user), conflict resolution (for collaborative environments), encryption (for sensitive data), and cloud synchronization (for cross-device access). This recommended approach has been battle-tested in production applications, balances competing concerns (performance vs. safety, simplicity vs. flexibility), and provides clear upgrade paths as requirements evolve.

6.4.7.1 Technology Stack

State Management: Zustand

```
import { create } from 'zustand';
import { persist, createJSONStorage } from 'zustand/middleware';
import { immer } from 'zustand/middleware/immer';

// Define store with TypeScript
interface SettingsStore {
  // State
  appearance: AppearanceSettings;
  behavior: BehaviorSettings;
  data: DataSettings;
  security: SecuritySettings;

  // Actions
  updateAppearance: (updates: Partial<AppearanceSettings>) => void;
  updateBehavior: (updates: Partial<BehaviorSettings>) => void;
  resetToDefaults: () => void;
  exportSettings: () => string;
  importSettings: (json: string) => void;
}

// Create store with middleware
const useSettingsStore = create<SettingsStore>()(  immer(
    persist(
      (set, get) => ({
        // Initial state
        appearance: {
          theme: 'light',
          fontSize: 14,
          sidebarVisible: true,
        },
        behavior: {
          autoSaveDelay: 30000,
          confirmOnDelete: true,
        },
        data: {
          refreshInterval: 60000,
          cacheEnabled: true,
        },
        security: {
          sessionTimeout: 3600000,
          twoFactorEnabled: false,
        },

        // Actions (using Immer for immutable updates)
        updateAppearance: (updates) => set((state) => {
          Object.assign(state.appearance, updates);
        }),
```

```
        updateBehavior: (updates) => set((state) => {
          Object.assign(state.behavior, updates);
        }),

        resetToDefaults: () => set(getDefaultSettings()),

        exportSettings: () => {
          const state = get();
          return JSON.stringify(state, null, 2);
        },

        importSettings: (json) => {
          const imported = JSON.parse(json);
          set(imported);
        },
      }),
      {
        name: 'settings-storage',
        storage: createJSONStorage(() => indexedDBStorage),
        partialize: (state) => ({
          // Only persist these fields
          appearance: state.appearance,
          behavior: state.behavior,
          data: state.data,
          security: state.security,
        }),
      }
    )
  )
);
```

Schema Validation: Zod

```
import { z } from 'zod';

// Define schemas with Zod
const AppearanceSettingsSchema = z.object({
  theme: z.enum(['light', 'dark', 'high-contrast']),
  fontSize: z.number().min(10).max(24),
  sidebarVisible: z.boolean(),
  gridDensity: z.enum(['compact', 'normal', 'comfortable']).default('normal'),
});

const BehaviorSettingsSchema = z.object({
  autoSaveDelay: z.number().min(1000).max(300000),
  confirmOnDelete: z.boolean(),
  undoHistorySize: z.number().min(10).max(100).default(50),
});

const DataSettingsSchema = z.object({
  refreshInterval: z.number().min(5000),
  cacheEnabled: z.boolean(),
```

```
  maxCacheSize: z.number().min(1024 * 1024).default(50 * 1024 * 1024),
});

const SecuritySettingsSchema = z.object({
  sessionTimeout: z.number().min(60000),
  twoFactorEnabled: z.boolean(),
  encryptionEnabled: z.boolean().default(true),
});

// Combined schema
const SettingsSchema = z.object({
  appearance: AppearanceSettingsSchema,
  behavior: BehaviorSettingsSchema,
  data: DataSettingsSchema,
  security: SecuritySettingsSchema,
  _version: z.number().optional(),
});

// Type inference
type Settings = z.infer<typeof SettingsSchema>;
type AppearanceSettings = z.infer<typeof AppearanceSettingsSchema>;

// Validation function
function validateSettings(settings: unknown): Settings {
  return SettingsSchema.parse(settings);
}
```

Persistence: Hybrid Strategy

```
class HybridPersistenceManager {
  private criticalSettings = ['security', 'data.credentials'];
  private debouncedSettings = ['appearance', 'behavior'];
  private debouncedSave = debounce(this.saveToIndexedDB.bind(this), 1000);

  async persist(category: string, data: any) {
    // Immediate persistence for critical settings
    if (this.isCritical(category)) {
      await this.saveToIndexedDB(category, data);
      console.log(`[Persistence] Immediate save: ${category}`);
      return;
    }

    // Debounced persistence for UI settings
    if (this.isDebounced(category)) {
      this.debouncedSave(category, data);
      console.log(`[Persistence] Debounced save queued: ${category}`);
      return;
    }

    // Default: immediate save
    await this.saveToIndexedDB(category, data);
  }
```

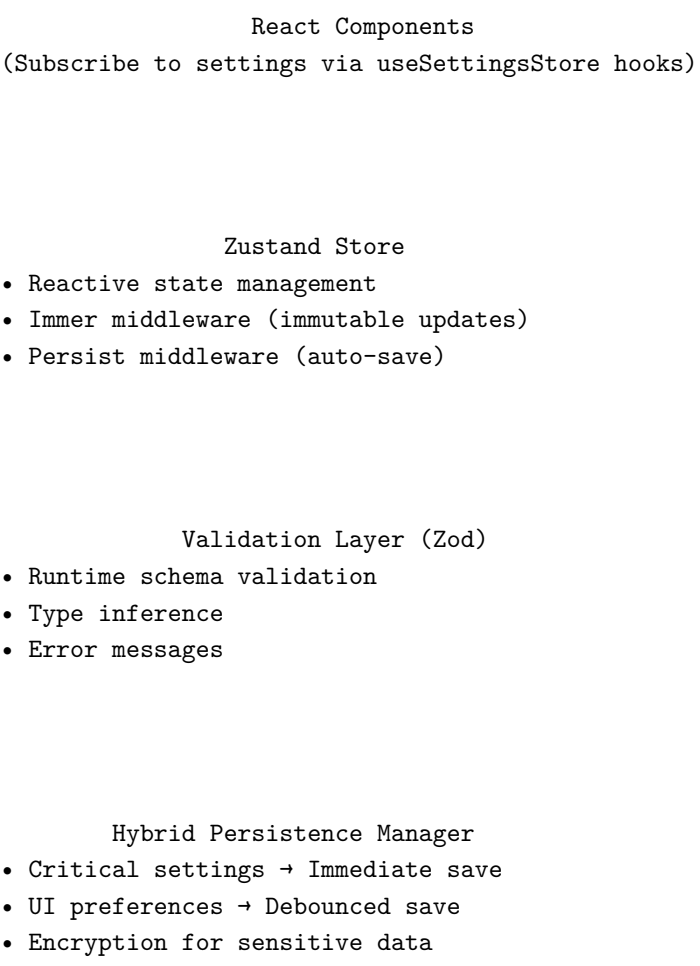
```
  private isCritical(category: string): boolean {
    return this.criticalSettings.some(c => category.startsWith(c));
  }

  private isDebounced(category: string): boolean {
    return this.debouncedSettings.some(c => category.startsWith(c));
  }

  private async saveToIndexedDB(category: string, data: any) {
    const db = await openDB('settings');
    await db.put('settings', { category, data, timestamp: Date.now() });
  }
}
```

Storage: IndexedDB

```
import { openDB, DBSchema } from 'idb';

// Define database schema
interface SettingsDB extends DBSchema {
  settings: {
    key: string;
    value: {
      category: string;
      data: any;
      timestamp: number;
    };
    indexes: { 'by-timestamp': number };
  };
  backups: {
    key: number;
    value: {
      settings: any;
      timestamp: number;
      version: number;
    };
  };
}

// Initialize database
const db = await openDB<SettingsDB>('settings-db', 1, {
  upgrade(db) {
    // Settings store
    const settingsStore = db.createObjectStore('settings', { keyPath: 'category' });
    settingsStore.createIndex('by-timestamp', 'timestamp');

    // Backups store
    db.createObjectStore('backups', { keyPath: 'timestamp' });
  },
});
```

```
// IndexedDB storage adapter for Zustand
const indexedDBStorage = {
  getItem: async (name: string) => {
    const item = await db.get('settings', name);
    return item?.data || null;
  },
  setItem: async (name: string, value: any) => {
    await db.put('settings', {
      category: name,
      data: value,
      timestamp: Date.now(),
    });
  },
  removeItem: async (name: string) => {
    await db.delete('settings', name);
  },
};
```

6.4.7.2 Architecture Diagram



```
IndexedDB                Cloud Sync
(Local)                   (Optional)
```

6.4.7.3 Implementation Checklist

Phase 1: Core Setup

- ☐ Install dependencies (zustand, zod, idb)
- ☐ Define TypeScript interfaces for all settings categories
- ☐ Create Zod schemas with validation rules
- ☐ Set up IndexedDB database with proper indexes
- ☐ Initialize Zustand store with persist middleware

Phase 2: Persistence

- ☐ Implement hybrid persistence manager
- ☐ Add debouncing for UI settings
- ☐ Add immediate persistence for critical settings
- ☐ Implement encryption for sensitive data
- ☐ Add error handling and retry logic

Phase 3: Migration

- ☐ Create migration manager
- ☐ Define migration functions for each version
- ☐ Implement rollback capabilities
- ☐ Add migration testing
- ☐ Set up backup system

Phase 4: Features

- ☐ Implement settings import/export
- ☐ Add settings reset functionality
- ☐ Create settings UI components
- ☐ Add cloud sync (optional)
- ☐ Implement audit logging

Phase 5: Optimization

- ☐ Add performance monitoring
- ☐ Optimize bundle size
- ☐ Implement lazy loading for settings categories
- ☐ Add compression for large settings
- ☐ Optimize IndexedDB queries

6.4.7.4 Why This Architecture?

Zustand over Redux

- Simpler API, less boilerplate
- No provider needed
- Excellent TypeScript support
- Smaller bundle size (~1KB vs ~8KB)
- Built-in persistence middleware

Zod over TypeScript-only

- Runtime validation catches invalid data
- Protects against corrupted storage
- Type inference reduces duplication
- Clear, actionable error messages
- Validates external data (imports, API responses)

IndexedDB over localStorage

- Much larger storage capacity (50MB-1GB vs 5-10MB)
- Asynchronous API (doesn't block UI)
- Supports complex queries with indexes
- Transaction support for data integrity
- Better performance for large datasets

Hybrid Persistence over Single Strategy

- Balances safety and performance
- Critical data saved immediately
- UI preferences don't cause excessive writes
- Reduces storage wear
- Better user experience (no lag on UI changes)

6.4.8 Core Components

The Settings Management system consists of several interconnected components that work together to provide a robust configuration infrastructure:

1. Settings Store (State Management)

- Central Zustand store holding all application settings in memory
- Provides reactive subscriptions for components to listen to changes
- Implements selectors for efficient partial state access
- Supports middleware for logging, persistence, and validation

2. Persistence Manager

- Coordinates saving settings to various storage backends
- Implements debouncing and batching for performance
- Handles storage quota management and cleanup
- Provides fallback mechanisms when storage is unavailable

3. Validation Engine

- Validates settings against Zod schemas before persistence
- Provides detailed error messages for invalid configurations
- Supports custom validation rules and business logic
- Ensures type safety at runtime

4. Sync Coordinator

- Manages synchronization between local and remote settings
- Implements conflict resolution strategies
- Handles offline scenarios with queue-based sync
- Provides real-time updates via WebSocket connections

5. Migration Manager

- Automatically migrates settings when schemas change
- Maintains version history for rollback capabilities
- Validates migrations before applying them
- Logs migration events for debugging

6. Encryption Service

- Encrypts sensitive settings using Web Crypto API
- Manages encryption keys securely
- Provides transparent encryption/decryption
- Supports key rotation for security

6.4.9 Settings Lifecycle

Settings flow through a well-defined lifecycle from initialization to persistence:

```
Initialize   → Load defaults, merge with stored settings

Validate     → Check against schemas, apply business rules

Update       → User changes settings via UI or API

Propagate    → Notify subscribers, trigger re-renders

Persist      → Save to IndexedDB, sync to cloud

Audit        → Log changes for compliance
```

Initialization Phase

1. Load default settings from configuration files
2. Retrieve stored settings from IndexedDB
3. Merge stored settings with defaults (stored takes precedence)
4. Apply organization-level policies and overrides
5. Validate merged settings against schemas
6. Initialize Zustand store with validated settings

Update Phase

1. User triggers setting change via UI component
2. Validation engine checks new value against schema
3. If valid, update in-memory store (optimistic update)

4. Trigger reactive subscriptions to notify components
5. Queue persistence operation (debounced or immediate)
6. If persistence fails, rollback in-memory change

Persistence Phase

1. Batch multiple pending changes if debounced
2. Serialize settings to JSON
3. Encrypt sensitive fields if configured
4. Write to IndexedDB with transaction
5. If cloud sync enabled, queue remote update
6. Log persistence event to audit trail

Synchronization Phase

1. Detect changes from remote source (other devices/users)
2. Fetch updated settings from server
3. Compare with local settings to detect conflicts
4. Apply conflict resolution strategy (last-write-wins, merge, prompt)
5. Update local store with resolved settings
6. Persist merged settings to IndexedDB

6.4.10 Conflict Resolution Strategies

When settings are modified concurrently across multiple devices or users, the system must resolve conflicts intelligently:

1. Last-Write-Wins (LWW)

- Simplest strategy: most recent change wins
- Uses timestamps to determine recency
- May lose user changes if not careful
- Best for: Single-user scenarios, non-critical settings

2. Merge Strategy

- Attempts to merge non-conflicting changes
- Uses operational transformation or CRDTs
- Preserves changes from both sources when possible
- Best for: Collaborative environments, structured settings

3. User Prompt

- Presents conflict to user for manual resolution
- Shows both versions side-by-side
- Allows user to choose or merge manually
- Best for: Critical settings, expert users

4. Field-Level Resolution

- Different strategies for different setting categories
- Critical settings use user prompt
- UI preferences use last-write-wins
- Data settings use merge strategy
- Best for: Production systems with mixed requirements

6.4.11 Performance Optimizations

Settings Management implements several optimizations to ensure minimal performance impact:

1. Selective Subscriptions

```
// Only re-render when specific settings change
const theme = useConfigStore((state) => state.appearance.theme);
const fontSize = useConfigStore((state) => state.appearance.fontSize);

// Avoid: subscribing to entire store
// const config = useConfigStore(); // Re-renders on ANY change
```

2. Debounced Persistence

```
const debouncedPersist = debounce((settings) => {
  persistToIndexedDB(settings);
}, 1000); // Wait 1s after last change

// User types in font size input: 12 → 13 → 14 → 15
// Only persists once after user stops typing
```

3. Lazy Loading

```
// Load settings on-demand rather than all at once
const loadSecuritySettings = async () => {
  if (!securitySettingsLoaded) {
    const settings = await fetchFromIndexedDB('security');
    setSecuritySettings(settings);
    securitySettingsLoaded = true;
  }
};
```

4. Compression

```
// Compress large configuration objects before storage
const compressed = await compress(JSON.stringify(settings));
await indexedDB.put('settings', compressed);

// Decompress on load
const compressed = await indexedDB.get('settings');
const settings = JSON.parse(await decompress(compressed));
```

5. Caching

```
// Cache frequently accessed settings in memory
const settingsCache = new Map();

function getSetting(key: string) {
  if (settingsCache.has(key)) {
    return settingsCache.get(key);
  }
  const value = store.getState()[key];
  settingsCache.set(key, value);
  return value;
}
```

6.4.12 Security Considerations

Settings Management implements multiple security layers to protect sensitive configuration data:

1. Encryption at Rest

- Sensitive settings (API keys, credentials) encrypted using AES-256-GCM
- Encryption keys derived from user password or stored in secure keychain
- Transparent encryption/decryption via middleware

2. Access Control

- Role-based permissions for modifying settings
- Organization admins can lock certain settings
- Audit trail for all setting changes

3. Input Validation

- All user input validated against schemas
- Prevents injection attacks via malicious settings
- Sanitizes string values before storage

4. Secure Transmission

- Settings synced over HTTPS only
- Certificate pinning for API requests
- JWT tokens for authentication

5. Content Security Policy

- Prevents XSS attacks via settings injection
- Restricts eval() and inline scripts
- Validates URLs in settings before navigation

6.4.13 Monitoring and Debugging

The Settings Management system provides comprehensive observability for troubleshooting and performance monitoring:

1. Audit Logging

```
interface SettingChangeEvent {
  timestamp: Date;
  userId: string;
  settingKey: string;
  oldValue: any;
  newValue: any;
  source: 'ui' | 'api' | 'sync' | 'migration';
}

// Log all changes
settingsStore.subscribe((state, prevState) => {
  const changes = detectChanges(prevState, state);
  changes.forEach(change => {
    auditLog.record({
      timestamp: new Date(),
      userId: currentUser.id,
      settingKey: change.key,
```

```
      oldValue: change.oldValue,
      newValue: change.newValue,
      source: 'ui',
    });
  });
});
```

2. Performance Metrics

- Track persistence latency (time to save settings)
- Monitor sync frequency and conflicts
- Measure validation overhead
- Alert on storage quota issues

3. Debug Mode

```
// Enable verbose logging in development
if (import.meta.env.DEV) {
  settingsStore.subscribe((state) => {
    console.log('[Settings] State updated:', state);
  });

  // Log all persistence operations
  persistenceManager.on('save', (key, value) => {
    console.log(`[Persistence] Saved ${key}:`, value);
  });
}
```

4. Settings Inspector

- DevTools panel showing current settings
- Visualize setting inheritance hierarchy
- Test setting changes without persistence
- Export/import for bug reproduction

6.4.14 Testing Strategies

Comprehensive testing ensures Settings Management reliability:

1. Unit Tests

- Test validation schemas with valid/invalid inputs
- Verify persistence manager debouncing logic
- Test conflict resolution algorithms
- Validate encryption/decryption roundtrips

2. Integration Tests

- Test full lifecycle: initialize → update → persist → sync
- Verify IndexedDB transactions and rollbacks
- Test migration from old to new schema versions
- Validate multi-tab synchronization

3. Performance Tests

- Benchmark persistence latency with large settings

- Test memory usage with many subscribers
- Measure sync overhead with high-frequency updates
- Validate storage quota handling

4. Security Tests

- Attempt to inject malicious values via settings
- Verify encryption keys are never logged
- Test access control enforcement
- Validate XSS prevention in setting values

6.5 Keybinding System

6.5.1 Concept

Customizable keyboard shortcuts for commands and actions.

6.5.2 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

6.5.3 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

Approach	Pros	Cons	Best For
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

6.5.4 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
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

6.5.5 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

6.5.6 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

6.5.7 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.

6.6 Theme System

6.6.1 Concept

Visual styling and color schemes that can be switched dynamically.

6.6.2 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

6.6.3 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
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

6.6.4 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

Library	Approach	Pros	Cons	Use Case
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
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

6.6.5 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

6.6.6 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

6.6.7 Recommended Architecture

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

6.7 Layout System

6.7.1 Concept

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

6.7.2 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>;
}
```

6.7.3 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

6.7.4 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

6.7.5 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: [/* ... */]
  }
};
```

6.7.6 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

6.7.7 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();
  }
}
```

6.7.8 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

6.7.9 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
```

6.8 Extension State

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

6.9 Storage Options & Strategy

6.9.1 Client-Side Storage

6.9.2 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

6.9.3 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

6.9.4 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

6.9.5 Server-Side/Hybrid Storage

6.9.5.1 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

6.9.5.2 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

6.9.6 Specialized Storage

6.9.6.1 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

6.9.6.2 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

6.9.7 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

6.9.8 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
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

Storage Type	Pros	Cons	Best For
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
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

6.9.9 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

Note: For detailed migration strategies including canvas state evolution, cell schema changes, version tracking, and user experience considerations, see the Migration Strategy section under User Configurations. 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
