tldraw Architecture Deep Dive - A Guide for Library Authors

Educational analysis of architectural patterns and design decisions

Generated through comprehensive source code analysis

October 30, 2025

Contents

Target Audience: Library authors seeking inspiration from tldraw's architecture

Purpose: Educational analysis of architectural patterns and design decisions

Note: This document focuses on understanding patterns for inspiration, not copying

---

1 Introduction

1.1 What is tldraw?

tldraw is an open-source infinite canvas SDK that powers collaborative whiteboarding applications. At its core, it is a sophisticated TypeScript library that provides a complete solution for building canvas-based editors with features like:

- Real-time collaborative editing
- Custom shapes and tools
- Infinite zoom and pan canvas
- Undo/redo with branching history
- Multiplayer synchronization
- Asset management and optimization
- Plugin architecture for extensions

What makes tldraw particularly interesting from an architectural perspective is not just what it does, but how it achieves high performance and maintainability while handling complex state management for thousands of interactive objects on a canvas.

1.2 Why This Document?

Modern web applications increasingly require sophisticated state management, real-time collaboration, and high-performance rendering. tldraw has solved many of these challenges in elegant ways that can inspire library authors working on similar problems.

This document is not a tutorial on using tldraw, nor is it encouraging you to copy tldraw's implementation. Instead, it aims to:

- Analyze the architectural decisions behind tldraw's design
- Explain the trade-offs and reasoning for choosing specific patterns
- Present alternative approaches you might consider
- Extract general principles applicable to other domains
- Guide you in making informed decisions for your own libraries

1.3 What Makes tldraw's Architecture Notable?

tldraw demonstrates several advanced architectural patterns:

1. Layered Architecture

tldraw is organized as a monorepo with clear separation of concerns across multiple packages, each building on the foundation of lower layers. This allows developers to use only what they need, from low-level reactive primitives up to a complete UI-enabled SDK.

2. Custom Reactive System

Rather than relying on existing state management solutions like Redux or MobX, tldraw built a custom signals-based reactive system optimized for their specific performance requirements. This decision reveals important insights about when custom solutions are justified.

3. Plugin Architecture

The ShapeUtil and StateNode patterns provide powerful extension points without forcing users to fork the codebase. This demonstrates thoughtful API design for libraries that need to be extensible.

4. Performance-First Design

With the need to handle 10,000+ shapes while maintaining 60fps, tldraw employs sophisticated optimization techniques like viewport culling, epoch-based dirty tracking, and hybrid data structures that adapt based on collection size.

5. Type Safety Throughout

Extensive use of TypeScript branded types, validators, and schema definitions ensures type safety across the entire stack, including serialized data and migrations.

1.4 Document Structure

This document is organized to progressively build understanding:

Chapters 1-3 examine the foundational packages (@tldraw/state, @tldraw/store, @tldraw/editor) that form the architectural core of tldraw.

Chapter 4 explores cross-cutting concerns like performance optimization, memory management, and type safety patterns that apply across multiple packages.

Chapter 5 presents alternative architectural approaches you might consider for similar problems, with analysis of trade-offs.

Chapter 6 distills key lessons for library authors, covering topics from API design to community building.

Chapter 7 provides a conclusion with key takeaways and further reading suggestions.

1.5 How to Read This Document

Depending on your interests, you might:

- **Read sequentially** if you want to understand the full architecture from bottom to top
- **Jump to specific chapters** if you're interested in particular concerns (e.g., Chapter 4 for performance patterns)
- **Focus on the "Alternative Approaches"** sections if you want to compare different architectural options
- **Read the "Lessons for Library Authors"** chapter for high-level principles without implementation details

Each section includes:

- Architectural overview explaining the design philosophy
- Source code analysis with real examples from tldraw
- Design decision rationale explaining why choices were made
- Alternative approaches you might consider
- Lessons applicable to your own projects

1.6 Prerequisites

This document assumes you have:

- Strong TypeScript/JavaScript knowledge
- Understanding of React fundamentals
- Familiarity with state management concepts
- Basic knowledge of reactive programming

You do not need prior experience with tldraw itself.

1.7 A Note on Inspiration vs. Copying

The patterns and architectures described here are educational resources. Your library should:

- Solve your specific problems, not tldraw's problems

- Make trade-offs appropriate to your constraints
- Consider your team's expertise and preferences
- Evaluate maintenance burden vs. benefits

tldraw's choices make sense for their requirements (high-performance canvas editor with real-time collaboration). Your requirements may lead to different, equally valid architectural decisions.

Let's begin by examining the foundation: tldraw's reactive signals architecture.

---

2 @tldraw/state - Reactive Signals Architecture

2.0.1 Architectural Overview

Core Philosophy: Fine-grained reactivity with automatic dependency tracking and lazy evaluation.

Key Insight: tldraw chose a signals-based approach (similar to SolidJS/MobX) over alternatives like Redux or React Context because:

- Sub-component re-rendering precision (only affected components update)
- Zero manual dependency arrays
- Better performance at scale (thousands of shapes)
- Framework agnostic (works beyond React)

2.0.2 Source Code Analysis

```
// packages/state/src/lib/types.ts
export interface Signal<Value, Diff = unknown> {
  name: string
  get(): Value
  lastChangedEpoch: number
  getDiffSince(epoch: number): RESET_VALUE | Diff[]
  __unsafe__getWithoutCapture(ignoreErrors?: boolean): Value
  children: ArraySet<Child>
}
```

2.0.2.1 1.1 The Signal Interface

Design Decision: The `Signal` interface is split into two implementations:

- **Atoms** (mutable): Direct state containers
- **Computed** (derived): Lazy, auto-updating derived values

Why this matters: This separation allows for clear data flow - atoms for sources of truth, computed for transformations.

```typescript
// packages/state/src/lib/Atom.ts (simplified)
class __Atom__<Value, Diff> implements Atom<Value, Diff> {
  private current: Value
  lastChangedEpoch = getGlobalEpoch()
  children = new ArraySet<Child>()
  historyBuffer?: HistoryBuffer<Diff>

  get(): Value {
    maybeCaptureParent(this)  // Automatic dependency tracking!
    return this.current
  }

  set(value: Value, diff?: Diff): Value {
    if (this.isEqual(this.current, value)) {
      return this.current  // No-op if unchanged
    }

    advanceGlobalEpoch()  // Increment global time

    // Store diff for history/undo
    if (this.historyBuffer) {
      this.historyBuffer.pushEntry(
        this.lastChangedEpoch,
        getGlobalEpoch(),
        diff ?? this.computeDiff(this.current, value)
      )
    }

    this.lastChangedEpoch = getGlobalEpoch()
    const oldValue = this.current
    this.current = value

    atomDidChange(this, oldValue)  // Notify children
    return value
  }
}
```

2.0.2.2 1.2 Atom Implementation Key Innovations:

1. **Epoch-Based Dirty Checking**: Instead of flag-based dirty tracking, uses a global incrementing counter
2. **Automatic Dependency Capture**: `maybeCaptureParent()` registers dependencies during `.get()` calls
3. **Diff Support**: Optional history tracking with custom diff functions
4. **Equality Optimization**: Skips updates if value hasn't actually changed

```ts
// packages/state/src/lib/Computed.ts (simplified)
class __UNSAFE__Computed<Value, Diff> implements Computed<Value, Diff> {
  private state: Value = UNINITIALIZED
  private lastCheckedEpoch = GLOBAL_START_EPOCH
  parentSet = new ArraySet<Signal<any, any>>()
  parents: Signal<any, any>[] = []
  children = new ArraySet<Child>()

  __unsafe__getWithoutCapture(ignoreErrors?: boolean): Value {
    const globalEpoch = getGlobalEpoch()

    // Short-circuit if value is still valid
    if (
      this.lastCheckedEpoch === globalEpoch ||
      !haveParentsChanged(this)
    ) {
      this.lastCheckedEpoch = globalEpoch
      return this.state
    }

    try {
      startCapturingParents(this)  // Begin tracking dependencies
      const result = this.derive(this.state, this.lastCheckedEpoch)
      const newState = result instanceof WithDiff ? result.value : result

      if (isUninitialized || !this.isEqual(newState, this.state)) {
        this.lastChangedEpoch = getGlobalEpoch()
        this.state = newState
      }

      this.lastCheckedEpoch = globalEpoch
      return this.state
    } finally {
      stopCapturingParents()  // End tracking
    }
  }

  get(): Value {
    const value = this.__unsafe__getWithoutCapture()
    maybeCaptureParent(this)  // Register this computed as a dependency
    return value
  }
}
```

2.0.2.3 1.3 Computed Implementation Key Innovations:

1. **Lazy Evaluation**: Only recomputes when accessed AND dependencies changed

2. **Memoization**: Returns cached value if parents haven't changed
3. **Incremental Computation**: Supports `withDiff()` for computing both value and diff simultaneously
4. **Error Resilience**: Errors reset state to `UNINITIALIZED` instead of crashing

```
// packages/state/src/lib/ArraySet.ts (simplified)
export class ArraySet<T> {
  private arraySize = 0
  private array: (T | undefined)[] | null = Array(8)  // Start as array
  private set: Set<T> | null = null  // Upgrade to Set later

  add(elem: T) {
    if (this.array) {
      const idx = this.array.indexOf(elem)
      if (idx !== -1) return false  // Already exists

      if (this.arraySize < 8) {
        this.array[this.arraySize] = elem
        this.arraySize++
        return true
      } else {
        // Upgrade to Set when threshold exceeded
        this.set = new Set(this.array)
        this.array = null
        this.set.add(elem)
        return true
      }
    }

    if (this.set) {
      if (this.set.has(elem)) return false
      this.set.add(elem)
      return true
    }
  }
}
```

2.0.2.4 1.4 ArraySet - Hybrid Data Structure Design Rationale:

- **Memory Optimization**: Arrays are more memory-efficient for small collections
- **Performance Trade-off**: Array operations are faster for ≤8 items, Set is faster for larger collections
- **Threshold Choice**: 8 was chosen based on profiling (typical dependency count per signal)

Inspiration for Your Library:

This pattern is excellent for managing:

- Event listener lists
- Observer patterns
- Dependency graphs
- Any collection that's usually small but occasionally large

2.0.3 Alternative Approaches

```
// Vue-inspired approach
function reactive<T extends object>(obj: T): T {
  return new Proxy(obj, {
    get(target, prop, receiver) {
      track(target, prop)  // Register dependency
      return Reflect.get(target, prop, receiver)
    },
    set(target, prop, value, receiver) {
      const result = Reflect.set(target, prop, value, receiver)
      trigger(target, prop)  // Notify dependents
      return result
    }
  })
}

// Usage
const state = reactive({ count: 0, name: 'John' })

const doubled = computed(() => state.count * 2)
```

2.0.3.1 Alternative 1: Proxy-Based Reactivity (Vue 3 style) **Pros**:

- More intuitive API (direct property access)
- No need for `.get()` / `.set()` methods
- Automatically reactive for nested objects

Cons:

- Proxy overhead on every property access
- Harder to optimize (can't use epoch-based checks)
- Doesn't work with primitives
- Trickier to debug (proxies hide implementation)

When to use:

- Object-heavy state (deeply nested structures)
- Developer experience is priority over raw performance
- Don't need to track changes to primitives directly

```
// RxJS-inspired approach
import { BehaviorSubject, combineLatest } from 'rxjs'
```

```
import { map } from 'rxjs/operators'

const count$ = new BehaviorSubject(0)
const name$ = new BehaviorSubject('John')

const greeting$ = combineLatest([count$, name$]).pipe(
  map(([count, name]) => `${name} has ${count} items`)
)

greeting$.subscribe(value => console.log(value))

count$.next(5)  // Triggers update
```

2.0.3.2 Alternative 2: Observable Streams (RxJS style) **Pros**:

- Rich operator library (debounce, throttle, etc.)
- Explicit data flow
- Time-based operations built-in
- Great for async/event streams

Cons:

- Steeper learning curve
- Subscription management overhead
- Not as performant for synchronous updates
- Verbose API

When to use:

- Heavy async operations
- Complex event streams
- Time-based transformations needed
- Already using RxJS ecosystem

```
// React hooks approach
function useDoubled(count: number) {
  return useMemo(() => count * 2, [count])  // Manual dependency array
}

function Counter() {
  const [count, setCount] = useState(0)
  const doubled = useDoubled(count)

  return <div>{doubled}</div>
}
```

2.0.3.3 Alternative 3: Manual Dependency Tracking (React style) **Pros**:

- Explicit dependencies (easier to reason about)

- No magic, clear data flow
- Framework-integrated (for React)
- Smaller bundle size

Cons:

- Manual dependency management (error-prone)
- Component-level granularity (over-renders)
- Framework-locked
- Verbose for complex dependencies

When to use:

- Already committed to React
- Simpler state requirements
- Team prefers explicit over automatic

2.0.4 Lessons for Library Authors

1. Choose Granularity Level

tldraw chose signal-level granularity. Consider:

- **Component-level** (React): Simpler, less precise
- **Signal-level** (tldraw): Complex, very precise
- **Object-level** (Proxy): Middle ground

2. Consider Performance Characteristics

tldraw's epoch-based approach means:

- O(1) dirty checking (compare two numbers)
- O(n) parent checking where n = number of dependencies
- O(1) update propagation start (just increment epoch)

3. Plan for Debugging

Notice tldraw's debug features:

- Every signal has a `name` field
- `__debug_ancestor_epochs__` for tracing
- `whyAmIRunning()` helper function
- Clear error messages

4. Memory Management Matters

ArraySet shows optimization for the 90% case:

- Profile your typical use case
- Optimize for common patterns
- Allow escape hatches for edge cases

2.0.5 Educational Implementation Exercise

Build a Mini Signals Library:

```
// Exercise: Implement this API
const count = atom('count', 0)
const doubled = computed('doubled', () => count.get() * 2)

const dispose = effect(() => {
  console.log('Count:', count.get())
  console.log('Doubled:', doubled.get())
})

count.set(5)  // Should log both values
dispose()     // Clean up
```

Key Concepts to Implement:

1. Global epoch counter
2. Dependency capture stack
3. Parent-child relationships
4. Lazy evaluation for computed
5. Effect scheduling

Hints:

- Use a global stack to track "current computation"
- Store parent signals in a Set for each computed/effect
- Compare epochs instead of dirty flags
- Only recompute if parent epochs have changed

---

3 @tldraw/store - Reactive Record Storage

3.0.1 Architectural Overview

Core Philosophy: Type-safe reactive database with automatic validation, migrations, and side effects.

Design Goals:

1. Each record in its own atom (fine-grained reactivity)
2. Schema-driven development (types from runtime definitions)
3. Built-in migration system
4. Side effects for business logic
5. Reactive queries/indexes

3.0.2 Source Code Analysis

```
// packages/store/src/lib/Store.ts (simplified)
export class Store<R extends UnknownRecord> {
  // Core reactive primitives
```

```
  private readonly records: AtomMap<IdOf<R>, R>  // Each record = 1 atom
  readonly history: Atom<number, RecordsDiff<R>>  // Change tracking
  readonly query: StoreQueries<R>                 // Reactive indexes
  readonly sideEffects: StoreSideEffects<R>       // Lifecycle hooks

  constructor(config: StoreConfig<R>) {
    this.schema = config.schema
    this.records = new AtomMap()
    this.history = atom('history', 0, { historyLength: 1000 })
    this.query = new StoreQueries(this.records, this.history)
    this.sideEffects = new StoreSideEffects(this)
  }

  get<K extends IdOf<R>>(id: K): RecordFromId<K> | undefined {
    return this.records.get(id)  // Reactive! Captures dependency
  }

  put(records: R[], source: ChangeSource = 'user'): void {
    const diff: RecordsDiff<R> = { added: {}, updated: {}, removed: {} }

    transact(() => {  // Atomic update
      for (const record of records) {
        // Validate
        const validated = this.schema.validate(record)

        // Run side effects
        const prev = this.records.get(record.id)
        if (!prev) {
          this.sideEffects.handleBeforeCreate(validated, source)
          this.records.set(record.id, validated)
          diff.added[record.id] = validated
          this.sideEffects.handleAfterCreate(validated, source)
        } else {
          this.sideEffects.handleBeforeChange(prev, validated, source)
          this.records.set(record.id, validated)
          diff.updated[record.id] = [prev, validated]
          this.sideEffects.handleAfterChange(prev, validated, source)
        }
      }

      // Update history
      this.history.update(n => n + 1, diff)
    })
  }
}
```

3.0.2.1 2.1 Store Architecture Key Design Decisions:

1. **AtomMap Instead of Single Atom**: Each record is individually reactive
2. **Transact Wrapper**: All updates are atomic
3. **Side Effects During Transaction**: Business logic runs in same transaction
4. **Diff Tracking**: History atom stores what changed

```
// packages/store/src/lib/AtomMap.ts (simplified)
export class AtomMap<Key extends string, Value> {
  private atoms = new Map<Key, Atom<Value>>()

  get(key: Key): Value | undefined {
    const atom = this.atoms.get(key)
    if (!atom) return undefined
    return atom.get()  // Reactive access!
  }

  set(key: Key, value: Value): void {
    let atom = this.atoms.get(key)
    if (!atom) {
      atom = atomLib.atom(`atom:${key}`, value)
      this.atoms.set(key, atom)
    }
    atom.set(value)  // Update triggers reactivity
  }

  delete(key: Key): void {
    const atom = this.atoms.get(key)
    if (atom) {
      this.atoms.delete(key)
      // Atom is now eligible for garbage collection
    }
  }
}
```

3.0.2.2 2.2 AtomMap - Reactive Map Why AtomMap?

- **Fine-grained updates**: Changing one record doesn't invalidate others
- **Memory efficient**: Atoms created lazily
- **Automatic cleanup**: Deleted records release memory
- **Map interface**: Familiar API

```
// packages/store/src/lib/StoreQueries.ts (simplified)
export class StoreQueries<R extends UnknownRecord> {
  private indexes = new Map<string, Computed<any>>()

  // Create reactive index by property
  index<T extends R['typeName'], K extends keyof R>(
```

```
    type: T,
    key: K
  ): Computed<Map<R[K], Set<IdOf<R>>>> {
    const indexKey = `${type}:${String(key)}`

    let index = this.indexes.get(indexKey)
    if (!index) {
      index = computed(indexKey, (prev, prevEpoch) => {
        // Incremental update using diffs!
        if (!isUninitialized(prev)) {
          const diff = this.history.getDiffSince(prevEpoch)
          if (diff !== RESET_VALUE) {
            return this.applyDiffToIndex(prev, diff, key)
          }
        }

        // Full rebuild
        const result = new Map<R[K], Set<IdOf<R>>>()
        for (const record of this.records.values()) {
          if (record.typeName === type) {
            const value = record[key]
            if (!result.has(value)) {
              result.set(value, new Set())
            }
            result.get(value)!.add(record.id)
          }
        }
        return withDiff(result, diff)  // Provide diff for next iteration
      }, { historyLength: 100 })

      this.indexes.set(indexKey, index)
    }

    return index
  }

  // Incremental index update
  private applyDiffToIndex(
    prevIndex: Map<any, Set<string>>,
    diff: RecordsDiff<R>,
    key: string
  ): Map<any, Set<string>> {
    const newIndex = new Map(prevIndex)  // Clone

    // Handle additions
    for (const record of Object.values(diff.added)) {
      const value = record[key]
```

```
      if (!newIndex.has(value)) {
        newIndex.set(value, new Set())
      }
      newIndex.get(value)!.add(record.id)
    }

    // Handle removals
    for (const record of Object.values(diff.removed)) {
      const value = record[key]
      const set = newIndex.get(value)
      if (set) {
        set.delete(record.id)
        if (set.size === 0) {
          newIndex.delete(value)  // Clean up empty sets
        }
      }
    }

    // Handle updates
    for (const [prev, curr] of Object.values(diff.updated)) {
      const prevValue = prev[key]
      const currValue = curr[key]

      if (prevValue !== currValue) {
        // Remove from old value's set
        newIndex.get(prevValue)?.delete(curr.id)
        // Add to new value's set
        if (!newIndex.has(currValue)) {
          newIndex.set(currValue, new Set())
        }
        newIndex.get(currValue)!.add(curr.id)
      }
    }

    return newIndex
  }
}
```

3.0.2.3 2.3 Reactive Queries Key Innovations:

1. **Incremental Updates**: Uses diffs to update indexes efficiently
2. **Lazy Index Creation**: Indexes only created when first accessed
3. **Automatic Invalidation**: Indexes recompute when records change
4. **Memory Management**: Empty sets are cleaned up

```
// packages/store/src/lib/StoreSideEffects.ts (simplified)
```

```
export class StoreSideEffects<R extends UnknownRecord> {
  private beforeCreate = new Map<string, Set<BeforeCreateHandler<R>>>()
  private afterCreate = new Map<string, Set<AfterCreateHandler<R>>>()
  private beforeChange = new Map<string, Set<BeforeChangeHandler<R>>>()
  private afterChange = new Map<string, Set<AfterChangeHandler<R>>>()

  registerAfterCreateHandler<T extends R>(
    typeName: T['typeName'],
    handler: (record: T, source: ChangeSource) => void
  ): void {
    if (!this.afterCreate.has(typeName)) {
      this.afterCreate.set(typeName, new Set())
    }
    this.afterCreate.get(typeName)!.add(handler as any)
  }

  handleAfterCreate(record: R, source: ChangeSource): void {
    const handlers = this.afterCreate.get(record.typeName)
    if (handlers) {
      for (const handler of handlers) {
        handler(record, source)
      }
    }
  }

  // Similar for beforeCreate, beforeChange, afterChange,
  // beforeDelete, afterDelete...
}
```

3.0.2.4 2.4 Side Effects System Use Cases for Side Effects:

1. **Cascade Updates**: Update related records
2. **Validation**: Prevent invalid state transitions
3. **Denormalization**: Maintain cached data
4. **External Sync**: Update external systems
5. **Audit Logging**: Track changes

3.0.3 Alternative Approaches

```
// Redux-inspired approach
interface StoreState {
  books: Record<string, Book>
  authors: Record<string, Author>
}

type Action =
  | { type: 'ADD_BOOK', book: Book }
```

```
  | { type: 'UPDATE_BOOK', id: string, changes: Partial<Book> }
  | { type: 'DELETE_BOOK', id: string }

function reducer(state: StoreState, action: Action): StoreState {
  switch (action.type) {
    case 'ADD_BOOK':
      return {
        ...state,
        books: {
          ...state.books,
          [action.book.id]: action.book
        }
      }
    case 'UPDATE_BOOK':
      return {
        ...state,
        books: {
          ...state.books,
          [action.id]: {
            ...state.books[action.id],
            ...action.changes
          }
        }
      }
    // etc.
  }
}
```

3.0.3.1 Alternative 1: Redux-Style Reducers Pros:

- Time-travel debugging
- Predictable state updates
- Easy to test
- Clear action log

Cons:

- Verbose (lots of boilerplate)
- Coarse-grained updates (entire store re-renders)
- Manual normalization required
- No fine-grained reactivity

When to use:

- Need time-travel debugging
- State history is critical
- Prefer explicit over automatic
- Don’t need sub-component optimization

```
// ORM-inspired approach
@Entity()
class Book {
  @PrimaryKey()
  id: string

  @Property()
  title: string

  @ManyToOne(() => Author)
  author: Author

  @BeforeCreate()
  validate() {
    if (!this.title) throw new Error('Title required')
  }

  @AfterUpdate()
  notifySubscribers() {
    // Notify interested parties
  }
}

const book = await em.findOne(Book, bookId)
book.title = 'New Title'
await em.flush()  // Persist changes
```

3.0.3.2 Alternative 2: ORM-Style (TypeORM/Sequelize) **Pros**:

- Familiar ORM patterns
- Declarative relationships
- Built-in lifecycle hooks
- Transaction support

Cons:

- Heavier runtime
- Async operations (not always desirable)
- Less control over reactivity
- Database-oriented (not in-memory focused)

When to use:

- Backend/database-heavy apps
- Team familiar with ORMs
- Need robust transaction support
- Persistence is primary concern

```typescript
// ECS-inspired approach (used in game engines)
class World {
  private entities = new Set<number>()
  private components = new Map<string, Map<number, any>>()

  createEntity(): number {
    const id = Math.random()
    this.entities.add(id)
    return id
  }

  addComponent<T>(entity: number, type: string, data: T): void {
    if (!this.components.has(type)) {
      this.components.set(type, new Map())
    }
    this.components.get(type)!.set(entity, data)
  }

  query(componentTypes: string[]): number[] {
    const entities: number[] = []
    for (const entity of this.entities) {
      if (componentTypes.every(type =>
        this.components.get(type)?.has(entity)
      )) {
        entities.push(entity)
      }
    }
    return entities
  }
}

// Usage
const world = new World()
const player = world.createEntity()
world.addComponent(player, 'position', { x: 0, y: 0 })
world.addComponent(player, 'velocity', { vx: 1, vy: 0 })

// Query all entities with position and velocity
const movingEntities = world.query(['position', 'velocity'])
```

3.0.3.3 Alternative 3: Entity-Component-System (ECS) **Pros**:

- Extreme flexibility
- High performance (cache-friendly)
- Composition over inheritance
- Great for dynamic systems

Cons:

- Unfamiliar to most developers
- Complex query system
- No type safety by default
- Overkill for simple CRUD

When to use:

- Game development
- Highly dynamic entity types
- Performance critical (thousands of entities)
- Need composition patterns

3.0.4 Lessons for Library Authors

1. Consider Granularity Trade-offs

tldraw chose per-record atoms:

- **Pro**: Surgical updates, minimal re-renders
- **Con**: More memory overhead, complex implementation

Alternatives:

- Per-collection atoms (simpler, less precise)
- Per-property atoms (most precise, highest overhead)

2. Design for Migration from Day 1

tldraw's migration system shows:

- Versions are inevitable
- Schema changes must be managed
- Forward/backward compatibility matters
- Test migrations thoroughly

3. Side Effects vs Pure Functions

tldraw uses side effects liberally:

- **Advantage**: Business logic co-located with data
- **Disadvantage**: Harder to test, potential for bugs

Alternative: Pure functions + separate effect system

4. Index Strategy

Consider your access patterns:

- **Frequent lookups by property**: Pre-build indexes
- **Rare queries**: Build on-demand
- **Write-heavy**: Lazy index updates
- **Read-heavy**: Eager index updates

3.0.5 Educational Implementation Exercise

Build a Mini Reactive Store:

```
// Exercise: Implement this API
interface Book {
  id: string
  title: string
  authorId: string
}

const store = createStore<Book>()

// Reactive get
const book = store.get('book:1')  // Should capture dependency

// Batch update
store.put([
  { id: 'book:1', title: '1984', authorId: 'author:1' },
  { id: 'book:2', title: 'Brave New World', authorId: 'author:2' }
])

// Reactive index
const booksByAuthor = store.query.index('authorId')
const authorBooks = computed(() => {
  return booksByAuthor.get().get('author:1') ?? new Set()
})

// Side effects
store.sideEffects.afterCreate((book) => {
  console.log('Book created:', book.title)
})
```

Key Concepts to Implement:

1. AtomMap for per-record storage
2. Transact wrapper for atomic updates
3. Reactive index with incremental updates
4. Side effect system with type-based handlers

---

4 @tldraw/editor - Core Editor Engine

4.0.1 Architectural Overview

Core Philosophy: Event-driven, state-machine based editor with pluggable shapes and tools.

Key Architecture Decisions:

1. **StateNode Pattern**: Tools are hierarchical state machines
2. **ShapeUtil Pattern**: Each shape type has a utility class
3. **Event System**: Centralized event processing
4. **Manager Pattern**: Separate concerns into specialized managers

4.0.2 Source Code Analysis

4.0.2.1 3.1 StateNode Architecture The StateNode system is tldraw's solution to complex tool interactions.

Why State Machines?

- Tool behavior depends on current state (idle vs dragging vs resizing)
- Clear transitions between states
- Easy to visualize and debug
- Hierarchical (tools have sub-states)

Example State Machine:

```
SelectTool (root)
├── Idle
├── Pointing
│   ├── PointingCanvas
│   ├── PointingShape
│   └── PointingHandle
├── Brushing
├── Translating
├── Resizing
└── Rotating
```

```typescript
// Simplified StateNode implementation
export abstract class StateNode {
  constructor(
    public editor: Editor,
    public parent?: StateNode
  ) {}

  // Lifecycle hooks
  onEnter(info: TLEnterEventInfo): void {}
  onExit(info: TLExitEventInfo): void {}

  // Event handlers
  onPointerDown(info: TLPointerEventInfo): void {}
  onPointerMove(info: TLPointerEventInfo): void {}
  onPointerUp(info: TLPointerEventInfo): void {}
  onKeyDown(info: TLKeyboardEventInfo): void {}

  // State transitions
  transition(id: string, info: any): void {
    const child = this.children[id]
```

```
    if (child) {
      this.current?.onExit(info)
      this.current = child
      child.onEnter(info)
    }
  }

  // Delegate to current child
  handleEvent(name: string, info: any): void {
    const handler = this.current?.[`on${name}`]
    if (handler) {
      handler.call(this.current, info)
    } else {
      // Bubble up to parent
      this.parent?.handleEvent(name, info)
    }
  }
}
```

Design Patterns in StateNode:

1. **Hierarchical State Pattern**: States can have sub-states
2. **Event Delegation**: Events bubble up if not handled
3. **Template Method**: Subclasses override lifecycle hooks
4. **Strategy Pattern**: Different states = different strategies

4.0.2.2 3.2 ShapeUtil Pattern

Every shape type needs:

- Geometry calculation
- Rendering
- Hit testing
- Resize/rotate handling
- Indicator (selection outline)

```
// Simplified ShapeUtil
export abstract class ShapeUtil<T extends TLShape> {
  abstract type: string

  // Required: Calculate geometric bounds
  abstract getGeometry(shape: T): Geometry2d

  // Required: Render the shape
  abstract component(shape: T): JSX.Element

  // Required: Render selection indicator
  abstract indicator(shape: T): JSX.Element

  // Optional: Handle resize
```

```
  onResize(shape: T, info: TLResizeInfo): T {
    return shape  // Default: no resize
  }

  // Optional: Handle rotation
  onRotate(shape: T, info: TLRotateInfo): T {
    return shape  // Default: no rotate
  }

  // Optional: Can bind to other shapes?
  canBind(shape: T): boolean {
    return false  // Default: no binding
  }

  // Helper: Check if point is inside shape
  hitTest(shape: T, point: VecLike): boolean {
    const geometry = this.getGeometry(shape)
    return geometry.hitTestPoint(point, 0, false)
  }
}
```

Why ShapeUtil Pattern?

- **Open/Closed Principle**: Add new shapes without modifying core
- **Single Responsibility**: Each shape's logic is isolated
- **Type Safety**: Generic type parameter ensures consistency
- **Reusability**: Common logic in base class

4.0.2.3 3.3 Editor Class

The Editor is the central coordinator:

```
// Extremely simplified Editor
export class Editor {
  // State
  readonly store: TLStore
  readonly user: UserPreferencesManager
  readonly history: HistoryManager

  // Managers
  readonly viewport: ViewportManager
  readonly camera: CameraManager
  readonly selection: SelectionManager
  readonly snap: SnapManager

  // Current tool
  private _currentTool: StateNode

  // API
  createShape(shape: Partial<TLShape>): this {
```

```
    const complete = this.schema.createShape(shape)
    this.store.put([complete])
    return this
  }

  updateShape(id: TLShapeId, changes: Partial<TLShape>): this {
    const shape = this.store.get(id)
    if (!shape) return this
    this.store.update(id, (s) => ({ ...s, ...changes }))
    return this
  }

  deleteShape(id: TLShapeId): this {
    this.store.remove([id])
    return this
  }

  // Selection
  select(...ids: TLShapeId[]): this {
    this.store.put([{
      ...this.instanceState,
      selectedShapeIds: ids
    }])
    return this
  }

  // Tool management
  setCurrentTool(id: string, info?: any): this {
    const tool = this.root.children[id]
    if (tool) {
      this._currentTool.onExit(info)
      this._currentTool = tool
      tool.onEnter(info)
    }
    return this
  }

  // Event dispatch
  dispatch(event: TLEventInfo): this {
    this._currentTool.handleEvent(event.name, event)
    return this
  }
}
```

Design Patterns in Editor:

1. **Facade Pattern**: Simple API hiding complex subsystems
2. **Manager Pattern**: Separate concerns (viewport, selection, etc.)

3. **Fluent Interface**: Method chaining for ergonomics
4. **Event Bus**: Centralized event dispatch

4.0.3 Alternative Approaches

```
// Command pattern for undo/redo
interface Command {
  execute(): void
  undo(): void
}

class CreateShapeCommand implements Command {
  constructor(
    private editor: Editor,
    private shape: TLShape
  ) {}

  execute(): void {
    this.editor.store.put([this.shape])
  }

  undo(): void {
    this.editor.store.remove([this.shape.id])
  }
}

class UpdateShapeCommand implements Command {
  constructor(
    private editor: Editor,
    private id: TLShapeId,
    private oldState: TLShape,
    private newState: TLShape
  ) {}

  execute(): void {
    this.editor.store.put([this.newState])
  }

  undo(): void {
    this.editor.store.put([this.oldState])
  }
}

// Usage
const history: Command[] = []
const cmd = new CreateShapeCommand(editor, shape)
cmd.execute()
```

```
history.push(cmd)

// Undo
history.pop()?.undo()
```

4.0.3.1 Alternative 1: Command Pattern **Pros**:

- Explicit command objects
- Easy to serialize/replay
- Clean undo/redo
- Audit trail built-in

Cons:

- More boilerplate
- Memory overhead (store old states)
- Complex for nested operations
- Not as direct as tldraw's approach

When to use:

- Need command serialization
- Complex undo/redo requirements
- Want explicit history
- Building collaborative editors

```
// Observer pattern without reactivity library
class ObservableStore<T> {
  private data = new Map<string, T>()
  private listeners = new Set<(id: string, value: T | undefined) => void>()

  get(id: string): T | undefined {
    return this.data.get(id)
  }

  set(id: string, value: T): void {
    this.data.set(id, value)
    this.notify(id, value)
  }

  delete(id: string): void {
    this.data.delete(id)
    this.notify(id, undefined)
  }

  subscribe(listener: (id: string, value: T | undefined) => void): () => void {
    this.listeners.add(listener)
    return () => this.listeners.delete(listener)
```

```
  }

  private notify(id: string, value: T | undefined): void {
    for (const listener of this.listeners) {
      listener(id, value)
    }
  }
}
```

4.0.3.2 Alternative 2: Observable Pattern (Direct) Pros:

- Simple implementation
- No external dependencies
- Explicit subscriptions
- Easy to debug

Cons:

- No fine-grained reactivity
- Manual subscription management
- No automatic dependency tracking
- Listeners get all changes

When to use:

- Simple applications
- Want minimal dependencies
- Don't need automatic tracking
- Few observers per observable

4.0.4 Lessons for Library Authors

1. State Machines for Complex Interactions

When to use state machines:

- Tool behavior varies by context
- Need clear state transitions
- Want to visualize flow
- Have hierarchical states

2. Plugin Architecture Patterns

tldraw's ShapeUtil shows:

- Define clear extension points
- Use abstract classes for shared logic
- Provide sensible defaults
- Make overrides optional

3. Manager Pattern for Separation

Split concerns into managers:

- Viewport/Camera management
- Selection management
- History/Undo management
- Snap/Guide management

Each manager:

- Has focused responsibility
- Can be tested in isolation
- Can be mocked/replaced
- Has clear dependencies

4. API Design

tldraw's fluent API:

```
editor
  .createShape({ type: 'geo' })
  .selectAll()
  .alignTop()
  .distributeHorizontally()
```

Principles:

- Chainable methods
- Predictable returns
- Batch operations
- Undo-able by default

---

5 Cross-Cutting Concerns

5.0.1 Performance Optimization Patterns

Pattern 1: Viewport Culling

```
// Only render visible shapes
const visibleShapes = computed('visibleShapes', () => {
  const viewport = editor.viewportPageBounds.get()
  const allShapes = editor.currentPage.shapes.get()

  return allShapes.filter(shape => {
    const bounds = editor.getShapePageBounds(shape)
    return bounds?.intersects(viewport)
  })
})
```

Pattern 2: Geometry Caching

```
// Cache expensive calculations
const geometryCache = new WeakCache<TLShape, Geometry2d>()
```

```
function getGeometry(shape: TLShape): Geometry2d {
  return geometryCache.get(shape, () => {
    return expensiveGeometryCalculation(shape)
  })
}
```

Pattern 3: Batch Updates

```
// Atomic multi-record updates
editor.batch(() => {
  shapes.forEach(shape => {
    editor.updateShape(shape.id, { x: shape.x + 10 })
  })
  // All updates happen atomically, UI updates once
})
```

5.0.2 Memory Management

Pattern 1: Weak References for Caches

```
// WeakCache for automatic cleanup
class WeakCache<K extends object, V> {
  private cache = new WeakMap<K, V>()

  get(key: K, factory: () => V): V {
    let value = this.cache.get(key)
    if (!value) {
      value = factory()
      this.cache.set(key, value)
    }
    return value
  }
}
```

Pattern 2: Subscription Cleanup

```
// Always clean up subscriptions
class MyComponent {
  private disposers: (() => void)[] = []

  mount() {
    this.disposers.push(
      store.listen(this.handleChange),
      editor.on('change', this.handleEditorChange)
    )
  }

  unmount() {
```

```
    this.disposers.forEach(dispose => dispose())
    this.disposers = []
  }
}
```

5.0.3 Type Safety Patterns

Pattern 1: Branded Types

```
// Prevent ID confusion
type TLShapeId = string & { __brand: 'ShapeId' }
type TLPageId = string & { __brand: 'PageId' }

function createShapeId(): TLShapeId {
  return `shape:${uniqueId()}` as TLShapeId
}

// TypeScript prevents mixing
const shapeId: TLShapeId = createShapeId()
const pageId: TLPageId = shapeId  // Error!
```

Pattern 2: Discriminated Unions

```
// Type-safe shape variants
type TLShape =
  | { type: 'geo'; props: { w: number; h: number; geo: GeoType } }
  | { type: 'text'; props: { text: string; size: TextSize } }
  | { type: 'arrow'; props: { start: VecLike; end: VecLike } }

function renderShape(shape: TLShape) {
  switch (shape.type) {
    case 'geo':
      return <GeoShape {...shape.props} />  // TypeScript knows props!
    case 'text':
      return <TextShape {...shape.props} />
    case 'arrow':
      return <ArrowShape {...shape.props} />
  }
}
```

---

6 Alternative Approaches

6.0.1 Architecture Alternatives

Option 1: CQRS (Command Query Responsibility Segregation)

Separate read and write models:

```
// Write side - Commands
class CreateShapeCommand {
  execute(store: WriteStore, shape: TLShape): void {
    store.append({ type: 'SHAPE_CREATED', shape })
  }
}

// Read side - Queries
class ShapeQueries {
  getShapeById(id: string): TLShape | undefined {
    return this.readModel.shapes.get(id)
  }

  getShapesByType(type: string): TLShape[] {
    return this.readModel.shapesByType.get(type) ?? []
  }
}

// Event sourcing
const events: Event[] = [
  { type: 'SHAPE_CREATED', shape: {...} },
  { type: 'SHAPE_UPDATED', id: '...', changes: {...} },
  { type: 'SHAPE_DELETED', id: '...' }
]
```

When to use:

- Need event sourcing
- Complex read models
- Audit trail critical
- Microservices architecture

Option 2: Actor Model

Each entity is an actor:

```
// Actor-based (inspired by Akka)
class ShapeActor {
  private state: TLShape

  async receive(message: Message): Promise<void> {
    switch (message.type) {
      case 'UPDATE':
        this.state = { ...this.state, ...message.changes }
        await this.notifyWatchers()
        break
      case 'DELETE':
        await this.cleanup()
        break
```

```
    }
  }
}

// Message passing
await shapeActor.send({ type: 'UPDATE', changes: {...} })
```

When to use:

- Distributed systems
- Concurrent operations
- Need isolation
- Erlang/Elixir background

6.0.2 Data Flow Alternatives

Option 1: Flux Architecture

```
// Unidirectional data flow
const actions = {
  createShape(shape: TLShape) {
    dispatcher.dispatch({ type: 'CREATE_SHAPE', shape })
  }
}

const dispatcher = {
  dispatch(action: Action) {
    stores.forEach(store => store.onDispatch(action))
  }
}

const store = {
  shapes: new Map(),
  onDispatch(action: Action) {
    switch (action.type) {
      case 'CREATE_SHAPE':
        this.shapes.set(action.shape.id, action.shape)
        this.emit('change')
        break
    }
  }
}
```

Option 2: Stream-Based

```
// RxJS-style streams
const shapeCreated$ = new Subject<TLShape>()
const shapeUpdated$ = new Subject<{ id: string; changes: Partial<TLShape> }>()
```

```
const allShapes$ = merge(
  shapeCreated$,
  shapeUpdated$.pipe(
    withLatestFrom(shapeStore$),
    map(([update, store]) => applyUpdate(store, update))
  )
)

allShapes$.subscribe(shapes => {
  renderShapes(shapes)
})
```

---

7 Lessons for Library Authors

7.0.1 Start with Architecture

Define Your Constraints:

1. **Performance Requirements**: How many entities? Update frequency?
2. **Developer Experience**: What API feels natural?
3. **Bundle Size**: Is size critical?
4. **Framework**: Agnostic or framework-specific?

tldraw's Constraints:

- Must handle 10,000+ shapes
- Sub-60fps updates required
- Framework agnostic (React is view layer)
- Developer experience is priority

7.0.2 Layer Your Abstractions

tldraw's layers:

```
@tldraw/tldraw (Complete SDK with UI)
      ↓
@tldraw/editor (Core engine, no shapes)
      ↓
@tldraw/store (Reactive storage)
      ↓
@tldraw/state (Reactive primitives)
```

Each layer:

- Has clear responsibility
- Can be used independently
- Builds on lower layers
- Hides implementation details

7.0.3 Design for Extension

Provide Extension Points:

1. **Custom Shapes**: ShapeUtil pattern
2. **Custom Tools**: StateNode pattern
3. **Custom UI**: Component overrides
4. **Custom Storage**: Store adapters
5. **Custom Sync**: Sync backends

Make Common Easy, Complex Possible:

```
// Easy: Use defaults
<Tldraw />

// Medium: Customize shapes
<Tldraw shapeUtils={[CustomShape]} />

// Advanced: Full control
<TldrawEditor
  store={customStore}
  shapeUtils={customShapes}
  tools={customTools}
  bindingUtils={customBindings}
/>
```

7.0.4 Invest in Developer Experience

tldraw's DX Investments:

1. **TypeScript**: Full type safety
2. **Documentation**: Comprehensive guides
3. **Examples**: 130+ working examples
4. **Error Messages**: Helpful, actionable
5. **Debugging Tools**: `whyAmIRunning()`, debug modes
6. **Testing Utilities**: `TestEditor` helper

7.0.5 Performance is a Feature

Profile Before Optimizing:

- tldraw profiled typical usage (100-1000 shapes)
- Optimized for 90% case (ArraySet threshold)
- Provided escape hatches (`__unsafe__getWithoutCapture`)

Optimize Strategically:

1. **Hot Paths**: Epoch comparison, geometry cache
2. **Memory**: ArraySet, WeakCache, cleanup
3. **Rendering**: Viewport culling, lazy evaluation
4. **Updates**: Batch transactions, diff-based updates

7.0.6 Plan for Scale

tldraw's Scalability Patterns:

1. **Fine-grained Reactivity**: Only affected components update
2. **Lazy Evaluation**: Computed values only when needed
3. **Incremental Updates**: Diffs instead of full recomputation
4. **Spatial Indexing**: R-tree for fast lookups
5. **Worker Threads**: Heavy calculations off main thread

7.0.7 Embrace Trade-offs

tldraw's Trade-offs:

- **Complexity for Performance**: Signals are harder than setState
- **Bundle Size for Features**: Full SDK is large (~500KB)
- **API Surface for Power**: Many methods, steep learning curve
- **Memory for Speed**: Caching, indexes, memoization

Document Your Decisions:

- Why signals over Redux?
- Why state machines for tools?
- Why per-record atoms?

7.0.8 Test Thoughtfully

tldraw's Testing Strategy:

1. **Unit Tests**: Core algorithms (geometry, math)
2. **Integration Tests**: Features (selection, copy/paste)
3. **E2E Tests**: User workflows (Playwright)
4. **Fuzz Tests**: Random operations
5. **Performance Tests**: Benchmark regressions

Test Utilities:

```
// Provide test helpers
class TestEditor extends Editor {
  createNShapes(n: number): TLShapeId[] {
    const ids: TLShapeId[] = []
    for (let i = 0; i < n; i++) {
      ids.push(this.createShape({ type: 'geo' }).selectedShapeIds[0])
    }
    return ids
  }

  clickShape(id: TLShapeId): this {
    const shape = this.getShape(id)!
    const center = this.getShapePageBounds(shape)!.center
    this.pointerDown(center.x, center.y)
```

```
    this.pointerUp(center.x, center.y)
    return this
  }
}
```

7.0.9 Document Architecture Decisions

Create ADRs (Architecture Decision Records):

```
# ADR-001: Use Signals for Reactivity

## Status
Accepted

## Context
Need fine-grained reactivity for 10,000+ shapes with 60fps updates.

## Decision
Implement custom signals library (@tldraw/state) instead of using Redux or MobX.

## Consequences
- **Positive**: Precise updates, better performance, framework agnostic
- **Negative**: Custom implementation, learning curve, maintenance burden
- **Mitigations**: Comprehensive tests, documentation, examples
```

7.0.10 Community and Adoption

Lower the Barrier:

1. **Quick Start**: `npm create tldraw` in 30 seconds
2. **Templates**: Vite, Next.js, Vue starters
3. **Examples**: Copy-paste ready code
4. **Discord**: Active community support
5. **Versioning**: Semantic versioning, migration guides

Listen to Users:

- GitHub issues (feature requests, bugs)
- Discord feedback (pain points, use cases)
- Analytics (which features are used)
- Case studies (how it's being used)

---

8 Conclusion

8.0.1 Key Takeaways

1. Architecture is About Trade-offs

tldraw chose:

- Complexity for performance
- Size for features
- Learning curve for power

Your library might choose differently based on your constraints.

2. Layered Abstractions Scale

Start with primitives (`@tldraw/state`), build up to features (`@tldraw/tldraw`).

3. Performance Requires Investment

Epoch-based tracking, ArraySet, viewport culling, geometry caching - these take time to implement but pay dividends.

4. Developer Experience Matters

Type safety, error messages, debugging tools, documentation - invest in DX from day one.

5. Extensibility is Freedom

ShapeUtil, StateNode, component overrides - allow users to extend your library without forking.

8.0.2 Further Reading

- **Signals**: SolidJS Reactivity
- **State Machines**: XState Documentation
- **ECS Pattern**: Entity Component System FAQ
- **Command Pattern**: Refactoring Guru - Command
- **Optimization**: Web Performance Working Group

8.0.3 Acknowledgments

This deep-dive was created by analyzing the tldraw source code as an educational resource for library authors. All credit for the architecture and implementation goes to the tldraw team.

tldraw Team:

- Steve Ruiz (@steveruizok)
- David Sheldrick (@ds300)
- Lu Wilson (@TodePond)
- And all contributors

Repository: https://github.com/tldraw/tldraw
Website: https://tldraw.dev
License: tldraw custom license (see repository)

Generated through comprehensive source code analysis of tldraw v3.15.1
Last Updated: October 30, 2025