Contents

1 tldraw Monorepo - Comprehensive Project Walkthrough

Generated: October 30, 2025

1.1 Project Overview

1.1.1 What is tldraw?

tldraw is a powerful infinite canvas SDK for building drawing, diagramming, and collaborative whiteboard applications in React. It provides both:

- **An open-source SDK** (`@tldraw/tldraw`) for developers to build custom canvas applications
- **A commercial whiteboard application** (tldraw.com) showcasing the SDK's capabilities

1.1.2 Repository Information

- **Repository Type**: TypeScript monorepo
- **Package Manager**: Yarn Berry (v4.7.0)
- **Build System**: LazyRepo (custom incremental build system)
- **Version**: 3.15.1 (across all packages)
- **Node.js Requirement**: ^20.0.0
- **React Requirement**: ^18.0.0 || ^19.0.0
- **License**: Custom tldraw license (business license available for watermark removal)

1.1.3 Key Features

- **Infinite canvas** with zoom, pan, and viewport management
- **Rich shape system** (arrows, text, drawings, geometric shapes, frames, notes, etc.)
- **Real-time collaboration** with WebSocket-based multiplayer sync
- **Reactive state management** using signals (similar to MobX/SolidJS)
- **Extensible architecture** for custom shapes, tools, and UI components
- **Asset management** for images, videos, and external content
- **Export capabilities** (SVG, PNG, JPEG, JSON)
- **Mobile-responsive** design with touch support
- **Internationalization** (40+ languages)

1.2 Repository Structure

1.2.1 Root Directory

```
tldraw-fork/
├── apps/                # Applications (examples, docs, tldraw.com, vscode)
├── packages/                # Core SDK packages
├── templates/               # Framework starter templates
├── internal/                # Internal development tools and scripts
├── assets/                  # Shared assets (fonts, icons, translations)
├── package.json             # Root package configuration
├── yarn.lock                # Dependency lock file
├── lazy.config.ts           # LazyRepo build configuration
├── eslint.config.mjs        # ESLint configuration
├── vitest.config.ts         # Vitest testing configuration
├── lerna.json               # Lerna configuration
└── tsconfig.json            # Root TypeScript configuration
```

1.2.2 Workspace Organization

The monorepo uses Yarn Berry workspaces with the following patterns:

```
{
  "workspaces": [
    "packages/*",
    "apps/*",
    "apps/vscode/*",
    "apps/dotcom/*",
    "internal/*",
    "templates/*"
  ]
}
```

1.3 Core Packages

1.3.1 Package Directory (`packages/`)

All core SDK packages are published to npm under the `@tldraw` namespace.

1.3.2 1. `@tldraw/editor` - Core Editor Engine

Location: `packages/editor/`
Purpose: Foundational infinite canvas editor without specific shapes or UI
Size: ~220 files, heavily TypeScript

Key Components:

- **Editor Class** (`Editor.ts`): Central orchestrator managing store, camera, selection, tools, and rendering
- **TldrawEditor Component** (`TldrawEditor.tsx`): React wrapper handling lifecycle and mounting
- **StateNode System**: Hierarchical finite state machine for tools
- **ShapeUtil System**: Abstract base classes for defining shape behavior
- **BindingUtil System**: Shape relationship management (e.g., arrows to shapes)
- **Managers**: Click, EdgeScroll, Focus, Font, History, Scribble, Snap, Text, Tick, UserPreferences

Architecture Highlights:

- Event-driven architecture using EventEmitter3
- Reactive state management via `@tldraw/state` signals
- Geometry primitives: `Vec`, `Mat`, `Box`, `Geometry2d` classes
- Text editing integration with Tiptap
- Export to SVG/PNG/JPEG/JSON

Usage Example:

```
import { TldrawEditor } from '@tldraw/editor'
import '@tldraw/editor/editor.css'

export default function MinimalEditor() {
```

```
  return (
    <TldrawEditor
      shapeUtils={customShapeUtils}
      tools={customTools}
      bindingUtils={customBindingUtils}
    />
  )
}
```

1.3.3 2. `@tldraw/tldraw` - Complete SDK with UI

Location: `packages/tldraw/`
Purpose: "Batteries included" SDK with full UI, tools, and shapes
Size: ~480 files (274 TS, 182 TSX)

What it Adds:

- **Default Shapes**: Text, Draw, Geo, Note, Line, Frame, Arrow, Highlight, Bookmark, Embed, Image, Video
- **Default Tools**: Select, Hand, Eraser, Laser, Zoom + shape creation tools
- **UI System**: Complete responsive UI with toolbar, menus, panels, dialogs
- **External Content Handlers**: Drag/drop files, paste URLs, bookmark unfurling
- **Arrow Bindings**: Smart arrow connections to shapes
- **Asset Pipeline**: Image/video upload, optimization, and management

Key Subsystems:

1. **TldrawUi Component**: Provider hierarchy for context, theming, translations
2. **Responsive Breakpoints**: Mobile, tablet, desktop adaptations
3. **SelectTool**: Sophisticated state machine with Idle, Brushing, Translating, Resizing, Rotating, Crop, EditingShape states
4. **Style Panel**: Color, size, opacity, dash, fill controls
5. **Minimap**: WebGL-accelerated canvas overview

Usage Example:

```
import { Tldraw } from 'tldraw'
import 'tldraw/tldraw.css'

export default function App() {
  return (
    <div style={{ position: 'fixed', inset: 0 }}>
      <Tldraw />
    </div>
  )
}
```

1.3.4 3. `@tldraw/state` - Reactive State Management

Location: `packages/state/`
Purpose: Fine-grained reactive signals library (similar to MobX/SolidJS)
Size: ~36 files

Core Concepts:

- **Atoms**: Mutable state containers
- **Computed**: Derived values with lazy evaluation
- **Effects**: Side effects with automatic dependency tracking
- **Transactions**: Atomic updates with rollback support
- **History**: Change tracking with diff-based updates

Architecture:

```
// Signal system
interface Signal<Value, Diff> {
  name: string
  get(): Value
  lastChangedEpoch: number
  getDiffSince(epoch: number): Diff[]
  children: ArraySet<Child>
}

// Core implementations
class Atom<Value, Diff> implements Signal<Value, Diff>
class Computed<Value, Diff> implements Signal<Value, Diff>
```

Key Features:

- Automatic dependency tracking during computation
- Epoch-based invalidation for efficient dirty checking
- `ArraySet` hybrid data structure for memory efficiency
- Custom equality and diff functions
- Pluggable effect scheduling (e.g., `requestAnimationFrame`)

Usage Example:

```
import { atom, computed, react } from '@tldraw/state'

const count = atom('count', 0)
const doubled = computed('doubled', () => count.get() * 2)
const stop = react('logger', () => console.log(doubled.get()))

count.set(5) // Logs: 10
```

1.3.5 4. `@tldraw/store` - Reactive Record Storage

Location: `packages/store/`
Purpose: Type-safe reactive database for managing record collections

Size: ~53 files

Core Components:

- **Store Class**: Central orchestrator for record management
- **AtomMap**: Reactive Map implementation (each record in its own atom)
- **RecordType System**: Factory for creating typed records with validation
- **StoreSchema**: Type registry with validation and migrations
- **StoreQueries**: Reactive indexing and query system
- **StoreSideEffects**: Lifecycle hooks (before/after create/update/delete)
- **Migration System**: Version-based schema evolution

Record Scopes:

- `document`: Persistent and synced across instances
- `session`: Per-instance, not synced but may be persisted
- `presence`: Per-instance, synced but not persisted (e.g., cursors)

Usage Example:

```
interface Book extends BaseRecord<'book'> {
  title: string
  author: IdOf<Author>
  numPages: number
}

const Book = createRecordType<Book>('book', {
  validator: bookValidator,
  scope: 'document',
}).withDefaultProperties(() => ({ numPages: 0 }))

const schema = StoreSchema.create({ book: Book, author: Author })
const store = new Store({ schema })

store.put([Book.create({ title: '1984', author: authorId })])
```

1.3.6 5. `@tldraw/tlschema` - Type Definitions & Validators

Location: `packages/tlschema/`
Purpose: Schema definitions, validators, and migrations for all tldraw data
Size: ~107 files (100 TypeScript)

Contents:

- Shape type definitions (TLArrowShape, TLTextShape, TLGeoShape, etc.)
- Asset types (TLImageAsset, TLVideoAsset, TLBookmarkAsset)
- Record types (TLPage, TLCamera, TLInstance, TLDocument)
- Style definitions (color, size, font, dash, fill, etc.)
- Validators using `@tldraw/validate`
- Migration sequences for schema evolution

1.3.7 6. `@tldraw/sync` & `@tldraw/sync-core` - Multiplayer System

Location: `packages/sync/` and `packages/sync-core/`
Purpose: Real-time collaboration infrastructure

Features:

- WebSocket-based real-time synchronization
- Conflict-free updates with operational transformation
- Presence awareness (cursors, selections)
- Connection state management with reconnection logic
- React hooks: `useSync()`, `useSyncDemo()`

Integration Example:

```
import { useSync } from '@tldraw/sync'

const store = useSync({
  uri: 'wss://demo.tldraw.xyz/connect/my-room',
  roomId: 'my-room',
})
```

1.3.8 7. `@tldraw/utils` - Shared Utilities

Location: `packages/utils/`
Purpose: Generic utilities used across all packages
Size: ~76 files

Categories:

- Math utilities (angles, vectors, curves)
- DOM utilities (measure text, canvas operations)
- Data structures (LRU cache, throttle, debounce)
- Type utilities (type guards, type helpers)
- Performance utilities (RAF loops, timers)

1.3.9 8. `@tldraw/validate` - Validation Library

Location: `packages/validate/`
Purpose: Lightweight Zod-inspired validation library

Features:

- Runtime type validation
- Type inference from validators
- Composable validators
- Used throughout tldraw for data integrity

1.3.10 9. `@tldraw/assets` - Asset Management

Location: `packages/assets/`
Purpose: Centralized asset distribution (fonts, icons, translations)

Contents:

- **Icons**: 161 SVG icons in sprite format
- **Fonts**: IBM Plex (Sans, Serif, Mono) + Shantell Sans
- **Translations**: 53 language files with regional variants
- **Embed Icons**: Service icons (YouTube, Figma, GitHub, etc.)

Export Strategies:

```
// Three distribution methods
import { getAssetUrlsByImport } from '@tldraw/assets/imports'
import { getAssetUrlsByMetaUrl } from '@tldraw/assets/urls'
import { getAssetUrls } from '@tldraw/assets/selfHosted'
```

1.3.11 10. Other Packages

- **`@tldraw/state-react`**: React hooks and components for state management
- **`@tldraw/dotcom-shared`**: Shared utilities for tldraw.com application
- **`@tldraw/worker-shared`**: Utilities for Cloudflare Workers
- **`@tldraw/fairy-shared`**: Shared code for fairy (AI) features
- **`@tldraw/namespaced-tldraw`**: Namespaced version of tldraw components
- **`@tldraw/create-tldraw`**: CLI tool (`npm create tldraw`) for project scaffolding

1.4 Applications

1.4.1 1. Examples App (`apps/examples/`)

Purpose: Primary development environment and SDK showcase
URL: https://examples.tldraw.com (production), localhost:5420 (dev)
Size: ~400 files (199 TSX, 139 MD)

Structure:

- 130+ examples demonstrating tldraw features
- Categories: getting-started, configuration, editor-api, ui, layout, events, shapes/tools, collaboration, data/assets, use-cases
- Each example: README.md with frontmatter + Component.tsx
- End-to-end tests using Playwright

Example Categories:

1. **Basic Usage**: `basic`, `readonly`, `hide-ui`
2. **API Integration**: `api`, `canvas-events`, `snapshots`
3. **Custom Shapes**: `custom-shape`, `ag-grid-shape`
4. **Custom Tools**: `custom-tool`, `lasso-select-tool`
5. **Collaboration**: `sync-demo`, `sync-custom-presence`
6. **Use Cases**: `pdf-editor`, `image-annotator`, `slides`

Development:

```
yarn dev  # Starts examples app at localhost:5420
yarn e2e  # Run end-to-end tests
```

1.4.2 2. Documentation Site (`apps/docs/`)

Purpose: Official documentation at https://tldraw.dev
Tech Stack: Next.js 15, SQLite, Algolia, Tailwind CSS
Size: ~150 files

Content System:

- **Human-written**: MDX files in `/content` (guides, tutorials)
- **Auto-generated**: API docs from TypeScript via Microsoft API Extractor
- **Database**: SQLite for content storage and search indexing
- **Search**: Algolia for full-text search

Build Process: 1. `fetch-api-source.ts` - Pulls TypeScript definitions 2. `create-api-markdown.ts` - Generates API docs via API Extractor 3. `refresh-content.ts` - Processes MDX and populates SQLite 4. `update-algolia-index.ts` - Updates search index

Content Sections:

- `getting-started/` - Quick start guides
- `docs/` - Core SDK documentation
- `reference/` - Auto-generated API reference
- `community/` - Contributing guides
- `blog/` - News and updates

Development:

```
yarn dev-docs  # Start docs development server
yarn refresh-api  # Regenerate API docs
yarn refresh-content  # Rebuild content database
```

1.4.3 3. tldraw.com Application (`apps/dotcom/`)

Purpose: Commercial collaborative whiteboard application
URL: https://tldraw.com

Architecture: Multi-workspace application with:

1.4.3.1 Client (`apps/dotcom/client/`) Tech Stack:

- React SPA with Vite
- Clerk for authentication
- React Router for routing
- FormatJS for i18n
- Sentry for error tracking

Size: ~472 files (122 TSX, 115 TS)

Key Features:

- User authentication and profile management
- File management (create, save, share, delete)
- Real-time collaboration with multiplayer sync
- Publishing and sharing functionality
- Responsive UI for mobile and desktop

Structure:

```
client/
├── src/
│   ├── tla/              # Main application code (124 files)
│   ├── fairy/            # AI features (62 files)
│   ├── components/       # Shared UI components
│   ├── hooks/            # React hooks
│   ├── utils/            # Utility functions
│   └── pages/            # Page components
├── public/               # Static assets
└── e2e/                  # End-to-end tests
```

1.4.3.2 Sync Worker (`apps/dotcom/sync-worker/`)

Purpose: Cloudflare Worker for multiplayer backend
Tech Stack: Cloudflare Workers + Durable Objects + PostgreSQL
Size: ~72 files (65 TS)

Responsibilities:

- Real-time collaboration via WebSocket
- File storage and retrieval
- PostgreSQL replication for data persistence
- User data synchronization
- Snapshot management

Key Durable Objects:

- `TLDrawDurableObject`: Room state and WebSocket connections
- `TLUserDurableObject`: User-specific data
- `TLLoggerDurableObject`: Logging and analytics
- `TLStatsDurableObject`: Statistics aggregation

1.4.3.3 Asset Upload Worker (`apps/dotcom/asset-upload-worker/`)

Purpose: Handle media uploads to Cloudflare R2
Size: Minimal (3 files)

Features:

- Image/video upload validation
- R2 object storage integration
- Asset deduplication
- Size and format constraints

1.4.3.4 Image Resize Worker (`apps/dotcom/image-resize-worker/`)

Purpose: Image optimization and format conversion
Size: Minimal (2 files)

Features:

- Dynamic image resizing
- Format conversion (AVIF, WebP, PNG, JPEG)
- CDN integration

1.4.3.5 Fairy Worker (`apps/dotcom/fairy-worker/`)

Purpose: AI-powered features
Size: ~16 files

Features:

- AI-assisted drawing
- Smart shape suggestions
- Natural language processing

1.4.3.6 Zero Cache (`apps/dotcom/zero-cache/`)

Purpose: Database synchronization layer
Tech Stack: Rocicorp Zero + PostgreSQL + PgBouncer

Features:

- Optimistic client-server sync
- PostgreSQL migrations (25 SQL files)
- Connection pooling with PgBouncer
- Docker setup for development

1.4.4 4. VSCode Extension (`apps/vscode/`)

Purpose: tldraw editor for `.tldr` files in VS Code
Size: Extension (25 files) + Editor (21 files)

Components:

1.4.4.1 Extension (`apps/vscode/extension/`)

- **Entry Point**: `src/extension.ts`
- **Editor Provider**: Custom editor for `.tldr` files
- **Webview Manager**: Communication with React webview
- **Document Handling**: File I/O for `.tldr` format

1.4.4.2 Editor (`apps/vscode/editor/`)

- React app rendering tldraw in webview
- RPC-based communication with extension
- File open/import UI
- Change tracking and synchronization

Features:

- Create and edit `.tldr` files
- Keyboard shortcuts (zoom, dark mode)
- Hot reload in development
- Compatible with tldraw.com files

Publishing:

- Automatic pre-releases on `main` branch
- Production releases on `production` branch
- Manual publishing via `yarn publish`

1.4.5 5. Bemo Worker (`apps/bemo-worker/`)

Purpose: Demo server for tldraw sync
URL: https://demo.tldraw.xyz
Tech Stack: Cloudflare Worker + Durable Objects + R2

Responsibilities: 1. **Asset Management**: Upload/retrieve assets to/from R2 2. **Bookmark Unfurling**: Extract metadata and save preview images 3. **Real-time Collaboration**: WebSocket connections to rooms

Environments:

- `dev`: Local development (port 8989)
- `preview`: Feature branch deployments
- `staging`: `canary-demo.tldraw.xyz`
- `production`: `demo.tldraw.xyz`

1.4.6 6. Analytics App (`apps/analytics/`)

Purpose: Unified analytics library with cookie consent
Output: Standalone JavaScript bundle (`analytics.js`)
Tech Stack: Vanilla TypeScript + Vite

Integrated Services:

- PostHog (product analytics + session recording)
- Google Analytics 4
- HubSpot (marketing automation)
- Reo (analytics service)

Features:

- Cookie consent management (GDPR/LGPD compliant)
- Geographic consent checking via CloudFlare worker
- Unified API: `identify()`, `reset()`, `track()`, `page()`
- UI components: Cookie banner, privacy settings dialog

Global API:

```
window.tlanalytics.identify('user-123', { plan: 'pro' })
window.tlanalytics.track('button_clicked', { button: 'upgrade' })
window.tlanalytics.page()
window.tlanalytics.openPrivacySettings()
```

1.4.7 7. Analytics Worker (`apps/analytics-worker/`)

Purpose: Geographic consent checking service
URL: `https://tldraw-consent.workers.dev`

Functionality:

- Detects user's country via CloudFlare `CF-IPCountry` header
- Returns whether explicit consent is required
- Countries requiring consent: EU, EEA, UK, Switzerland, Brazil

---

1.5 Templates

1.5.1 Template Directory (`templates/`)

Starter templates for different frameworks and use cases.

1.5.2 1. Framework Templates

1.5.2.1 Vite Template (`templates/vite/`)

- Fastest way to get started
- Minimal setup with hot module replacement
- Single HTML file + React component

1.5.2.2 Next.js Template (`templates/nextjs/`)

- Server-side rendering support
- App Router integration
- Production-ready configuration

1.5.2.3 Vue Template (`templates/vue/`)

- Vue 3 integration
- TSX wrapper component
- Vite build setup

1.5.3 2. Multiplayer Templates

1.5.3.1 Sync Cloudflare (`templates/sync-cloudflare/`)

- Complete multiplayer implementation
- Cloudflare Durable Objects backend

- Asset upload handling
- Bookmark unfurling

1.5.3.2 Simple Server Example (`templates/simple-server-example/`)

- Basic Node.js multiplayer server
- WebSocket communication
- Simple file storage

1.5.3.3 Socket.io Server Example (`templates/socketio-server-example/`)

- Socket.io-based multiplayer
- Easy to deploy
- Real-time synchronization

1.5.4 3. Advanced Use Case Templates

1.5.4.1 Agent Template (`templates/agent/`)

- AI agent integration
- Chat panel with history
- Custom tools and actions
- Cloudflare Worker backend with Durable Objects
- **Size**: ~138 files (85 TS, 43 TSX)

Features:

- Todo list management
- Context-aware prompts
- Target area and shape tools
- Wikipedia integration
- Action system with 26+ actions

1.5.4.2 Workflow Template (`templates/workflow/`)

- Node-based visual programming
- Executable workflows
- Custom node types (input, output, operation, condition)
- Connection validation
- Execution engine

Size: ~52 files (37 TSX)

1.5.4.3 Branching Chat Template (`templates/branching-chat/`)

- AI-powered conversational UI
- Node-based chat trees
- Branch visualization
- Custom AI worker integration

1.5.4.4 Chat Template (`templates/chat/`)

- Simple chat interface
- Message history
- User avatars
- Next.js-based

1.5.4.5 Shader Template (`templates/shader/`)

- WebGL shader integration
- Custom shape with GLSL shaders
- Interactive config panel
- Multiple shader examples (minimal, rainbow, shadow, fluid)

1.6 Internal Tools & Infrastructure

1.6.1 Internal Directory (`internal/`)

Development tools, scripts, and configuration.

1.6.2 1. Config Package (`internal/config/`)

Purpose: Shared configuration for monorepo

Contents:

- `tsconfig.base.json`: Base TypeScript configuration for all packages
- `api-extractor.json`: API documentation generation settings
- `vitest/setup.ts`: Global test setup with Canvas mocking
- `vitest/node-preset.ts`: Node.js Vitest configuration

1.6.3 2. Scripts (`internal/scripts/`)

Purpose: Build, deployment, and maintenance automation
Size: ~65 files (50 TS, 8 shell scripts)

Key Scripts:

1.6.3.1 Build & Package Management

- `build-api.ts`: Generate API documentation via API Extractor
- `build-package.ts`: Build individual packages
- `check-packages.ts`: Validate package configurations
- `typecheck.ts`: Type check all packages
- `api-check.ts`: Validate public API consistency

1.6.3.2 Publishing

- `publish-new.ts`: Publish new SDK version
- `publish-patch.ts`: Publish patch release
- `publish-prerelease.ts`: Publish pre-release
- `publish-manual.ts`: Manual publishing workflow
- `publish-vscode-extension.ts`: Publish VS Code extension

1.6.3.3 Deployment

- `deploy-analytics.ts`: Deploy analytics app
- `deploy-bemo.ts`: Deploy bemo worker
- `deploy-dotcom.ts`: Deploy tldraw.com
- `prune-preview-deploys.ts`: Clean up old preview deployments

1.6.3.4 Asset Management

- `refresh-assets.ts`: Rebuild and bundle assets
- `license-report.ts`: Generate license report
- `purge-css.ts`: Remove unused CSS

1.6.3.5 Internationalization

- `i18n-upload-strings.ts`: Upload translation strings
- `i18n-download-strings.ts`: Download translations

1.6.3.6 Context & Documentation

- `context.ts`: Find and display CONTEXT.md files
- `refresh-context.ts`: Update CONTEXT.md files using Claude
- `update-pr-template.ts`: Update PR template

1.6.3.7 Version Management

- `bump-versions.ts`: Bump package versions
- `extract-draft-changelog.tsx`: Generate changelog

1.6.4 3. Dev Tools (`internal/dev-tools/`)

Purpose: Git bisect helper for debugging

Features:

- Find PR causing specific issue
- Automated bisect process
- PR information display

1.6.5 4. Health Worker (`internal/health-worker/`)

Purpose: Updown.io webhook → Discord alert forwarding
Tech Stack: Cloudflare Worker

Features:

- Monitor service health
- Send Discord notifications
- Track uptime events

1.6.6 5. Apps Script (`internal/apps-script/`)

Purpose: Google Apps Script for Meet integration
Contents: Google Workspace app configuration

1.7 Build System & Configuration

1.7.1 LazyRepo Build System

Configuration: `lazy.config.ts`

LazyRepo is a custom incremental build system optimized for monorepos:

Features:

- **Incremental builds**: Only rebuild changed packages
- **Dependency resolution**: Automatic workspace dependencies
- **Intelligent caching**: File-based cache invalidation
- **Parallel execution**: Where dependencies allow
- **Top-level commands**: Run scripts across entire repo

Key Scripts Configuration:

```
{
  scripts: {
    build: {
      baseCommand: 'exit 0',
      runsAfter: {
        prebuild: {},
        'refresh-assets': {},
        'build-eslint-plugin': {},
        'build-i18n': {},
      },
      workspaceOverrides: {
        'packages/*': {
          runsAfter: { 'build-api': { in: 'self-only' } },
          cache: { inputs: ['api/**/*', 'src/**/*'] },
        },
      },
```

```
    },
    dev: {
      execution: 'independent',
      runsAfter: { predev: {}, 'refresh-assets': {} },
      cache: 'none',
    },
    'build-types': {
      execution: 'top-level',
      baseCommand: 'tsx internal/scripts/typecheck.ts',
      cache: {
        inputs: ['**/*.{ts,tsx}', 'tsconfig.json'],
        outputs: ['**/*.tsbuildinfo'],
      },
    },
  },
}
```

Cache Configuration:

```
baseCacheConfig: {
  include: [
    '<rootDir>/package.json',
    '<rootDir>/yarn.lock',
    '<rootDir>/lazy.config.ts',
    '<rootDir>/internal/config/**/*',
    'package.json',
  ],
  exclude: [
    '**/coverage/**/*',
    '**/dist*/**/*',
    '**/.next*/**/*',
    '**/*.tsbuildinfo',
  ],
}
```

1.7.2 TypeScript Configuration

Base Config: `internal/config/tsconfig.base.json`

Settings:

- Strict mode enabled
- Composite builds for workspace references
- Declaration generation for API docs
- React JSX configuration
- Vitest globals included

Workspace References: Packages use workspace references for incremental compilation:

```
{
  "extends": "config/tsconfig.base.json",
  "references": [
    { "path": "../utils" },
    { "path": "../store" }
  ]
}
```

1.7.3 ESLint Configuration

File: `eslint.config.mjs`

Custom Rules:

- `local/no-export-star`: Prevent `export *` usage
- `local/no-internal-imports`: Prevent internal imports
- `local/tagged-components`: Enforce component naming
- `local/prefer-class-methods`: Prefer class methods over arrow functions
- `local/tsdoc-param-matching`: Validate TSDoc parameter names
- `local/no-whilst`: Prevent `whilst` usage
- `local/no-fairy-imports`: Restrict fairy imports (dotcom only)

Workspace-Specific Rules:

1. **Core Packages** (`editor`, `tldraw`, `utils`):
   - No direct `fetch` usage (use `@tldraw/util` wrapper)
   - No direct `Image` usage
   - No direct timers (use `editor.timers`)
   - Require `referrerPolicy` on `<img>` tags
2. **Examples** (`apps/examples`):
   - Relaxed syntax rules for clarity
   - No `@internal` API usage
3. **TLA** (`apps/dotcom/client/src/tla`):
   - No string literals in JSX (i18n enforcement)
   - Must use `useIntl` from utils
4. **Templates**:
   - Relaxed rules for educational purposes
   - Console logs allowed

1.7.4 Vitest Configuration

Root Config: `vitest.config.ts`

```
import glob from 'glob'
import { defineConfig } from 'vitest/config'

const vitestPackages = glob.sync('./{apps,packages}/**/vitest.config.ts')

export default defineConfig({
```

```
  test: { projects: vitestPackages },
})
```

Test Setup: `internal/config/vitest/setup.ts`

Includes:

- Canvas mocking (vitest-canvas-mock)
- Animation frame polyfills
- Text encoding utilities
- WebCrypto polyfills
- Custom Jest matchers

1.7.5 Prettier Configuration

Package: `prettier ^3.6.1`
Plugin: `prettier-plugin-organize-imports` for auto-import sorting

Scripts:

```
yarn format  # Format all files
yarn format-current  # Format changed files only
```

1.7.6 Linter Hooks

Husky: Pre-commit hooks configured
lint-staged: Runs Prettier on staged files

```
{
  "lint-staged": {
    "*.{js,jsx,ts,tsx,json}": [
      "prettier --write --cache --log-level=warn"
    ]
  }
}
```

1.8 Development Workflow

1.8.1 Getting Started

```

# Clone repository
git clone https://github.com/tldraw/tldraw
cd tldraw

# Enable corepack for Yarn Berry
npm i -g corepack
```

```
# Install dependencies
yarn

# Build all packages (first time)
yarn build
```

1.8.1.1 Initial Setup

1.8.1.2 Development Commands Main Development:

```
# Start examples app (primary development environment)
yarn dev  # Runs at localhost:5420

# Start tldraw.com client
yarn dev-app

# Start documentation site
yarn dev-docs

# Start VS Code extension development
yarn dev-vscode

# Run specific template
yarn dev-template <template-name>
```

Building:

```
# Build all packages
yarn build

# Build SDK packages only
yarn build-package

# Build tldraw.com client
yarn build-app

# Build documentation site
yarn build-docs

# Build API documentation
yarn build-api

# Type check all packages
yarn build-types  # or yarn typecheck
```

Code Quality:

```
# Lint all packages
yarn lint
```

```
# Format code
yarn format

# Format only changed files
yarn format-current

# Type check
yarn typecheck

# API consistency check
yarn api-check

# Check package configurations
yarn check-packages
```

1.8.2 Working with Examples

Examples are the primary development environment for the SDK.

Location: `apps/examples/src/examples/`

Creating a New Example:

1. Create folder: `src/examples/my-example/`
2. Add `README.md` with frontmatter:

---

title: My Example
component: ./MyExample.tsx
category: editor-api
priority: 10
keywords: [example, feature]

One-line summary

Detailed description of what this example demonstrates.

Code explanation

Explain the key concepts with [1], [2] style footnotes.

---

[1] Explanation of concept 1 [2] Explanation of concept 2

3. Create component: `MyExample.tsx`

```
import { Tldraw } from 'tldraw'
import 'tldraw/tldraw.css'

export default function MyExample() {
```

```
  return (
    <div className="tldraw__editor">
      <Tldraw />
    </div>
  )
}
```

Guidelines:

- Use numbered footnote comments: `// [1]`, `// [2]`
- Keep "tight" examples minimal and focused
- Add realistic UI for "use-case" examples
- External CSS should match example name
- See `apps/examples/writing-examples.md` for details

1.8.3 Working with Packages

Development Pattern:

```
# Navigate to specific package
cd packages/editor

# Run package-specific tests
yarn test run

# Run with filter
yarn test run --grep "selection"

# Watch mode
yarn test

# Build this package only
lazy run build --filter 'packages/editor'
```

IMPORTANT: Run tests from the package directory, not repo root!

1.8.4 Asset Management

Assets Location: `/assets` and `/apps/dotcom/client/assets`

After Modifying Assets:

```
yarn refresh-assets  # Rebuild and bundle all assets
```

Asset Types:

- Icons: SVG sprite format
- Fonts: WOFF2 format
- Translations: JSON format
- Embed icons: PNG format

1.8.5 Context Files

tldraw uses AI-friendly `CONTEXT.md` files throughout the repository.

Find Context:

```
# Show nearest CONTEXT.md path
yarn context

# Show full content
yarn context -v

# Find all CONTEXT.md files
yarn context -r

# For specific file/directory
yarn context ./path/to/file
```

Refresh Context (requires Claude Code CLI):

```
yarn refresh-context
```

1.9 Testing Strategy

1.9.1 Testing Frameworks

1. **Vitest**: Unit and integration tests
2. **Playwright**: End-to-end tests
3. **jest-matcher-utils**: Custom matchers

1.9.2 Vitest Tests

Test File Naming:

- Unit tests: `ComponentName.test.ts` alongside source
- Integration tests: `src/test/feature-name.test.ts`

Running Tests:

```
# From package directory
cd packages/editor
yarn test run  # Run all tests once
yarn test      # Watch mode

# Filter tests
yarn test run --grep "selection"

# Coverage
yarn test-coverage  # From repo root
```

Test Structure:

```
import { TestEditor } from './test/TestEditor'

describe('MyFeature', () => {
  let editor: TestEditor

  beforeEach(() => {
    editor = new TestEditor()
  })

  it('should do something', () => {
    editor.createShape({ type: 'geo', id: ids.box1 })
    expect(editor.getOnlySelectedShape()).toBe(editor.getShape(ids.box1))
  })
})
```

Testing Best Practices:

- Test in tldraw workspace if you need shapes/tools
- Use `TestEditor` for editor testing
- Mock external dependencies
- Keep tests focused and fast

1.9.3 Playwright E2E Tests

Examples E2E (`apps/examples/e2e/`):

```
yarn e2e           # Run examples E2E tests
yarn e2e-ui        # Run with Playwright UI
```

Dotcom E2E (`apps/dotcom/client/e2e/`):

```
yarn e2e-dotcom    # Run dotcom E2E tests
yarn e2e-dotcom-x10  # Run 10 times
```

Test Structure:

```
// Fixtures
class Editor {
  async clickToolbarItem(name: string) { /* ... */ }
  async createShape(type: string) { /* ... */ }
}

// Tests
test.describe('Editor', () => {
  test('should create shapes', async ({ page }) => {
    const editor = new Editor(page)
    await editor.clickToolbarItem('rectangle')
    await editor.createShape('geo')
  })
```

```
})
```

1.9.4 Test Coverage

```
yarn test-coverage  # Generate coverage report
```

Opens coverage report in browser after generation.

1.10 Asset Management

1.10.1 Asset Pipeline Architecture

Central Assets: `/assets` directory - `fonts/`: IBM Plex fonts + Shantell Sans - `icons/icon/`: 161 SVG icons - `translations/`: 53 language JSON files - `embed-icons/`: Service icons (18 PNG files)

Dotcom Assets: `/apps/dotcom/client/assets` - Application-specific icons (57 SVG files)

1.10.2 Asset Processing

Refresh Pipeline:

```
yarn refresh-assets  # Triggered by internal/scripts/refresh-assets.ts
```

Process: 1. Collect assets from source directories 2. Optimize SVGs with SVGO 3. Bundle into packages 4. Generate type definitions 5. Create asset manifests

1.10.3 Asset Distribution

Three Export Strategies:

```
// 1. Imports (webpack/vite import URLs)
import { getAssetUrlsByImport } from '@tldraw/assets/imports'
const urls = getAssetUrlsByImport()

// 2. URLs (meta.url based)
import { getAssetUrlsByMetaUrl } from '@tldraw/assets/urls'
const urls = getAssetUrlsByMetaUrl()

// 3. Self-hosted (provide base URL)
import { getAssetUrls } from '@tldraw/assets/selfHosted'
const urls = getAssetUrls({ baseUrl: 'https://cdn.example.com' })
```

Usage in Editor:

```
import { Tldraw } from 'tldraw'
import { getAssetUrlsByImport } from '@tldraw/assets/imports'
```

```tsx
const assetUrls = getAssetUrlsByImport()

export default function App() {
  return <Tldraw assetUrls={assetUrls} />
}
```

1.10.4 Dynamic Assets (tldraw.com)

Upload Pipeline: 1. Client uploads to Asset Upload Worker 2. Worker validates size/type 3. Storage in Cloudflare R2 4. Hash-based deduplication 5. Optional image optimization via Image Resize Worker

Supported Formats:

- Images: PNG, JPEG, GIF, WebP, AVIF, SVG
- Videos: MP4, WebM
- Size limits enforced per type

1.10.5 External Content Handling

Bookmark Unfurling:

```http
// Server-side
POST /bookmarks/unfurl
{
  "url": "https://example.com"
}

// Response
{
  "url": "https://example.com",
  "title": "Example Site",
  "description": "Description text",
  "image": "https://example.com/preview.png"
}
```

Embed Integration: Supported services: YouTube, Figma, Excalidraw, GitHub Gist, Google Maps, Spotify, Vimeo, etc.

1.11 Architecture Deep Dive

1.11.1 Reactive State System

tldraw's architecture is built on a signals-based reactive state management system.

Core Concept:

```
Signal = Atom (mutable) or Computed (derived)
```

```
↓
Dependency Tracking (automatic)
↓
Efficient Updates (epoch-based)
↓
Effects & Reactions (side effects)
```

Example Flow:

```typescript
// 1. Create mutable state
const selectedIds = atom('selectedIds', new Set<string>())

// 2. Derive computed values
const selectedShapes = computed('selectedShapes', () => {
  const ids = selectedIds.get()
  return Array.from(ids).map(id => store.get(id))
})

// 3. React to changes
react('update-ui', () => {
  const shapes = selectedShapes.get()
  updateUI(shapes)
})

// 4. Mutation triggers cascade
selectedIds.set(new Set(['shape1', 'shape2']))
// Automatically: selectedShapes recomputes → reaction runs
```

Optimization:

- Lazy evaluation: Computed values only recalculated when needed
- Epoch comparison: Fast dirty checking
- Automatic cleanup: No memory leaks
- Transactions: Batch updates

1.11.2 Store Architecture

Three-Layer System:

```
TLStore (document data)
↓
AtomMap (reactive records)
↓
Atoms (individual records)
```

Record Flow:

```typescript
// Create
store.put([{ id: 'shape1', type: 'geo', /* ... */ }])

// AtomMap creates Atom<TLShape>
```

```
// → Side effects run (beforeCreate, afterCreate)
// → History tracked
// → Reactive queries update

// Update
store.update('shape1', (shape) => ({ ...shape, x: 100 }))

// Same Atom updated with new value
// → Side effects run (beforeChange, afterChange)
// → Diff calculated
// → History accumulated
// → Dependent computed values invalidate

// Query
const geoShapes = computed('geo-shapes', () => {
  return store.query.records('shape').filter(s => s.type === 'geo')
})

// Reactive index automatically maintained
// → Only recomputes when relevant records change
```

1.11.3 Shape System Architecture

ShapeUtil Pattern:

```
class MyShapeUtil extends ShapeUtil<TLMyShape> {
  static override type = 'my-shape' as const

  // Required: Geometric representation
  getGeometry(shape: TLMyShape) {
    return new Rectangle2d({ width: shape.props.w, height: shape.props.h })
  }

  // Required: React component for rendering
  component(shape: TLMyShape) {
    return <div style={{ width: shape.props.w, height: shape.props.h }}>
      {/* Custom rendering */}
    </div>
  }

  // Required: Selection indicator
  indicator(shape: TLMyShape) {
    return <rect width={shape.props.w} height={shape.props.h} />
  }

  // Optional: Handle interactions
  onResize(shape: TLMyShape, info: TLResizeInfo) {
```

```
    return { ...shape, props: { ...shape.props, w: info.newW, h: info.newH } }
  }
}
```

Registration:

```
<TldrawEditor shapeUtils={[MyShapeUtil]} />
```

1.11.4 Tool State Machine Architecture

StateNode Hierarchy:

```
RootState
├── SelectTool
│   ├── Idle
│   ├── Brushing
│   ├── Translating
│   ├── Resizing
│   ├── Rotating
│   └── EditingShape
├── GeoTool
│   ├── Idle
│   └── Pointing
├── HandTool
└── EraserTool
```

Tool Implementation:

```
export class MyTool extends StateNode {
  static override id = 'my-tool'

  // Lifecycle
  onEnter() {
    this.editor.setCursor({ type: 'cross', rotation: 0 })
  }

  onExit() {
    this.editor.setCursor({ type: 'default', rotation: 0 })
  }

  // Events
  onPointerDown(info: TLPointerEventInfo) {
    const { point } = info
    this.editor.createShape({
      id: createShapeId(),
      type: 'my-shape',
      x: point.x,
      y: point.y,
    })
```

```
  }

  onKeyDown(info: TLKeyboardEventInfo) {
    if (info.key === 'Escape') {
      this.editor.setCurrentTool('select')
    }
  }
}
```

1.11.5 Event Flow

Complete Event Cascade:

```
1. DOM Event (click, keydown, wheel)
   ↓
2. Event Managers (PointerManager, KeyboardManager)
   ↓
3. Editor Event Processing
   ↓
4. Current Tool StateNode
   ↓
5. Tool Handler (onPointerDown, onKeyDown, etc.)
   ↓
6. Editor State Update
   ↓
7. Store Transaction
   ↓
8. Reactive Updates
   ↓
9. Component Re-render
```

Example:

```
// User clicks canvas
1. <canvas> onClick
2. PointerManager captures event
3. Editor.dispatch({ type: 'pointer', name: 'pointer_down', ... })
4. SelectTool.onPointerDown(info)
5. SelectTool transitions to Brushing state
6. Editor.setSelectedShapes(...)
7. Store.put([{ id: 'instance', selectedShapeIds: [...] }])
8. selectedShapes computed invalidates
9. SelectionForeground re-renders
```

1.11.6 Multiplayer Sync Architecture

Sync Flow:

```
Local Editor
```

```
↓ (change)
TLStore.listen()
↓ (diff)
TLSyncClient.sendMessage()
↓ (WebSocket)
Server (TLDrawDurableObject)
↓ (broadcast)
Remote Clients
↓ (apply diff)
TLStore.mergeRemoteChanges()
↓ (update)
Remote Editor
```

Conflict Resolution:

- Operational transformation on server
- Client applies remote changes as 'remote' source
- Side effects don't run for remote changes
- Presence data (cursors) separate from document data

Presence System:

```
// Send presence
editor.updateInstanceState({ cursor: { x: 100, y: 200 } })

// Receive presence
const presences = usePresence()
presences.forEach(presence => {
  renderCursor(presence.cursor)
})
```

1.11.7 UI Architecture (TldrawUi)

Provider Hierarchy:

```
TldrawUi
└── TldrawUiContextProvider
    ├── TooltipProvider
    ├── TranslationProvider
    ├── EventProvider
    ├── DialogProvider
    ├── ToastsProvider
    └── BreakpointProvider
        └── Layout Components
```

Component Override System:

```
<Tldraw
  components={{
    Toolbar: MyCustomToolbar,
    SharePanel: MySharePanel,
```

```
    MenuPanel: null,  // Hide menu
  }}
/>
```

Responsive Breakpoints:

- Mobile: < 640px
- Tablet: 640px - 1024px
- Desktop: > 1024px

Layout Zones:

- Top: Menu, Helper Buttons, Top Panel, Share/Style Panels
- Bottom: Navigation, Toolbar, Help Menu
- Canvas: Main drawing area
- Overlays: Dialogs, Toasts, Minimap

1.11.8 Performance Optimizations

1. Viewport Culling: Only render shapes in viewport + small buffer

```
const shapesInViewport = editor.getCurrentPageShapes().filter(shape => {
  return editor.getShapePageBounds(shape)?.intersects(viewportBounds)
})
```

2. Geometry Caching:

```
// Geometry calculated once and cached
const geometry = editor.getShapeGeometry(shape)
// Cache invalidated when shape changes
```

3. Reactive Optimization:

```
// Avoid creating dependencies for metadata reads
const processShape = computed('process', () => {
  const shape = shapeAtom.get()  // Dependency
  const metadata = unsafe__withoutCapture(() => metadataAtom.get())  // No dependency
  return expensiveOperation(shape, metadata)
})
```

4. Batch Updates:

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

5. WebGL Minimap: Minimap uses WebGL for efficient rendering of large canvases

6. Asset Optimization:

- SVG sprite sheets for icons
- WOFF2 font subsetting
- Image lazy loading
- Hash-based deduplication

1.11.9 Memory Management

Automatic Cleanup:

```
// Signals clean up when no more children
const myComputed = computed('my-computed', () => {
  return expensiveCalculation()
})

// When last dependent is removed, myComputed garbage collected

// Explicit cleanup
const stop = react('my-reaction', () => {
  doSomething()
})
stop()  // Remove reaction
```

Store History Pruning:

```
// Configure history length
const store = new Store({
  schema,
  props: {
    historyLength: 100,  // Keep last 100 changes
  },
})
```

1.11.10 Error Handling

Error Boundaries:

- React error boundaries catch render errors
- Graceful degradation for failed external content
- Sentry integration for production error tracking

Validation Pipeline:

```
// Schema validation on every record
const validator = T.object({
  id: T.string,
  type: T.literal('my-shape'),
  props: T.object({
    w: T.number.positive(),
    h: T.number.positive(),
  }),
```

```
})

// Validation errors logged but don't crash
```

Safe Migrations:

```
// Migrations wrapped in try-catch
// Fallback to last known good version on error
```

1.12 Conclusion

The tldraw monorepo is a sophisticated, production-ready infinite canvas SDK with:

- **Strong architecture**: Reactive state, extensible shapes/tools, modular design
- **Production apps**: tldraw.com, VS Code extension, documentation site
- **Developer experience**: Comprehensive examples, templates, documentation
- **Collaboration**: Real-time multiplayer with conflict resolution
- **Performance**: Optimized rendering, caching, lazy evaluation
- **Type safety**: Full TypeScript with strict checking
- **Testing**: Unit, integration, and E2E test coverage
- **Build system**: Incremental builds with intelligent caching
- **Asset management**: Centralized distribution with multiple strategies

The codebase demonstrates professional software engineering practices including monorepo management, reactive architecture, extensibility patterns, and comprehensive tooling. It serves as both a powerful SDK for developers and a complete reference implementation through tldraw.com.

For More Information:

- Website: https://tldraw.dev
- Repository: https://github.com/tldraw/tldraw
- Examples: https://examples.tldraw.com
- Discord: https://discord.tldraw.com
- Twitter: @tldraw

Getting Started:

```
npm create tldraw@latest
# or
npx @tldraw/create-tldraw@latest
```

This documentation was generated through comprehensive exploration of the tldraw monorepo structure, source code, and configuration files.