

Sou9: Zero-Latency Commerce Through a Fully Client-Side Architecture

Technical Whitepaper · Principal Frontend Architecture · October 2026

Stack: Next.js 16 (App Router, Turbopack) · TypeScript · Tailwind CSS · Zustand · Framer Motion · Recharts

1. Abstract

Conventional commerce stacks place the network on the critical path of nearly every interaction: add-to-cart, price display, filter, checkout, dashboard render. Each hop (CDN, edge, API gateway, service, database) contributes latency variance, failure modes and operational surface area. Sou9 inverts this model. It is a complete commerce platform (storefront, cart, checkout flow and administrative analytics console) that executes **100% client-side**, with no backend and no database.

The architectural thesis is that for a large class of commerce workloads (catalog browsing, cart mutation, order review, operational reporting), the authoritative working set is small enough to live in the browser, and the browser is a sufficiently capable runtime to own it.

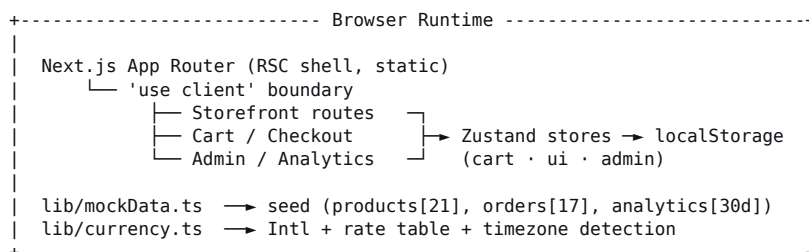
Key properties of the platform:

- **Zero network round-trips on the interaction path.** Every state transition resolves synchronously in memory, then persists to `localStorage`.
- **Three Zustand stores** (`cart`, `ui`, `admin`) form the entire state layer.
- **Deterministic seed data** in `lib/mockData.ts`: **21 products, 17 orders and 30 days of analytics**.
- **Custom currency module** supporting **16 currencies** with timezone-based auto-detection.
- **Static-exportable deployment surface.** The artifact is HTML, JS and CSS served from any CDN, with no servers to patch, scale or secure.

This paper documents the architecture, the persistence and hydration strategy, the design-token and motion systems, the analytics aggregation pipeline, and the explicit trade-offs of the approach.

2. Architecture: Zustand and LocalStorage Hydration

2.1 System Topology



Next.js 16 with Turbopack provides the build and dev toolchain and the App Router's file-system routing. The server component layer is reduced to a static shell; all meaningful application logic sits behind `'use client'` boundaries. There are no Route Handlers, Server Actions or database drivers in the dependency graph.

2.2 Store Decomposition

State is partitioned by **lifecycle and ownership**, not by feature. Three stores keep subscription graphs narrow and persistence policies independent.

Store	Responsibility	Persisted	Mutation frequency
cart	Line items, quantities, derived totals	Yes	High
ui	Drawer/modal state, selected currency, theme, filters	Partial (preferences only)	High
admin	Products, orders, analytics window, admin session flag	Yes	Low

Design rules:

- **No cross-store imports of state.** Stores communicate through selectors consumed in components, never by reading each other inside actions. This prevents hidden coupling and circular initialization.
- **Derived data is computed, not stored.** Totals, counts and aggregates are selectors or memoized hooks. Storing derived values invites drift.
- **Actions are pure transitions.** Each action is a function of previous state and arguments, with no I/O other than the persistence middleware.

2.3 The Cart Store

```
// stores/cart.ts
import { create } from 'zustand';
import { persist, createJSONStorage } from 'zustand/middleware';

export interface CartLine {
  productId: string;
  quantity: number;
  unitPriceBase: number; // minor units, base currency
}

interface CartState {
  lines: CartLine[];
  add: (productId: string, unitPriceBase: number, qty?: number) => void;
  setQty: (productId: string, qty: number) => void;
  remove: (productId: string) => void;
  clear: () => void;
}

export const useCartStore = create<CartState>()(
  persist(
    (set) => ({
      lines: [],
      add: (productId, unitPriceBase, qty = 1) =>
        set((s) => {
          const existing = s.lines.find((l) => l.productId === productId);
          return existing
            ? {
                lines: s.lines.map((l) =>
                  l.productId === productId
                    ? { ...l, quantity: l.quantity + qty }
                    : l
                ),
              }
            : { lines: [...s.lines, { productId, quantity: qty, unitPriceBase }] };
        }),
      setQty: (productId, qty) =>
        set((s) => ({
          lines:
            qty <= 0
              ? s.lines.filter((l) => l.productId !== productId)
              : s.lines.map((l) =>
                  l.productId === productId ? { ...l, quantity: qty } : l
                ),
        })),
      remove: (productId) =>
        set((s) => ({ lines: s.lines.filter((l) => l.productId !== productId) })),
      clear: () => set({ lines: [] }),
    })),
  ),
);
```

```

    {
      name: 'sou9:cart',
      version: 1,
      storage: createJSONStorage(() => localStorage),
      skipHydration: true, // hydrated manually after mount (see 2.4)
      partialize: (s) => ({ lines: s.lines }), // never persist functions
    }
  )
);

// Derived selectors: not persisted, not stored
export const selectItemCount = (s: CartState) =>
  s.lines.reduce((n, l) => n + l.quantity, 0);

export const selectSubtotalBase = (s: CartState) =>
  s.lines.reduce((sum, l) => sum + l.quantity * l.unitPriceBase, 0);

```

Money is stored in base-currency minor units (integers). Display conversion happens at the edge (Section 2.6), which eliminates floating-point drift in cart arithmetic and makes currency switching a pure view concern.

2.4 Hydration Strategy

Server-rendered HTML cannot know `localStorage` contents. Reading persisted state during the first render produces a **hydration mismatch** (server markup says 0 items, client says 3). Sou9 resolves this with deferred, explicit rehydration:

```

// hooks/useHydrated.ts
'use client';
import { useEffect, useState } from 'react';
import { useCartStore } from '@stores/cart';
import { useUIStore } from '@stores/ui';
import { useAdminStore } from '@stores/admin';

export function useHydrated(): boolean {
  const [ready, setReady] = useState(false);

  useEffect(() => {
    let cancelled = false;
    Promise.all([
      useCartStore.persist.rehydrate(),
      useUIStore.persist.rehydrate(),
      useAdminStore.persist.rehydrate(),
    ]).then(() => {
      if (!cancelled) setReady(true);
    });
    return () => {
      cancelled = true;
    };
  }, []);

  return ready;
}

// components/CartBadge.tsx
'use client';
export function CartBadge() {
  const hydrated = useHydrated();
  const count = useCartStore(selectItemCount);
  // Render a stable placeholder until hydrated, so SSR and first client paint match
  return <span aria-live='polite'>{hydrated ? count : 0}</span>;
}

```

Properties of this approach:

- **First paint is deterministic.** The server and the initial client render both emit the empty-state markup.
- **Rehydration runs once, post-mount.** The stores then update and React reconciles only the subscribed leaves.
- **Failure is non-fatal.** If `localStorage` is unavailable (private mode, quota, disabled storage), `rehydrate()` resolves with defaults and the app operates in memory-only mode.

2.5 Seed Data and Schema Versioning

`lib/mockData.ts` is the single source of truth for initial catalog and operational data:

```
// lib/mockData.ts (shape)
export interface Product {
  id: string;
  name: string;
  category: string;
  priceBase: number; // minor units
  stock: number;
  image: string;
}

export interface Order {
  id: string;
  createdAt: string; // ISO 8601, UTC
  status: 'pending' | 'paid' | 'shipped' | 'delivered' | 'cancelled';
  lines: { productId: string; quantity: number; unitPriceBase: number }[];
  totalBase: number;
}

export interface DailyMetric {
  date: string; // YYYY-MM-DD
  revenueBase: number;
  orders: number;
  visitors: number;
}

export const products: Product[]; // 21 records
export const orders: Order[]; // 17 records
export const analytics: DailyMetric[]; // 30 daily records
```

The `admin` store initializes from this seed **only when no persisted state exists**. Persisted mutations (edited stock, changed order status) then supersede the seed. Schema evolution is handled by the `persist` middleware's `version` and `migrate` hooks:

```
persist(creator, {
  name: 'sou9:admin',
  version: 2,
  migrate: (persisted: any, fromVersion) => {
    if (fromVersion < 2) {
      // v1 -> v2: introduce order.status enum, default legacy records to 'paid'
      persisted.orders = persisted.orders.map((o: any) => ({
        status: 'paid',
        ...o,
      }));
    }
    return persisted;
  },
});
```

This is the client-side equivalent of a database migration, executed lazily on the user's device at next load. Versioned keys (`sou9:cart`, `sou9:admin`, `sou9:ui`) also permit a global reset path (`localStorage.removeItem`) for the admin 'restore demo data' action.

2.6 Currency Module

The currency module is a pure-TypeScript layer with no network dependency. It provides:

- **A registry of 16 currencies** keyed by ISO 4217 code, each with locale hint, minor-unit exponent and a static rate relative to the base currency.
- **Timezone auto-detection**, mapping the IANA timezone reported by the runtime to a default currency on first visit.
- ****Formatting via Intl.NumberFormat****, so grouping, symbol placement and decimal rules come from the platform rather than hand-rolled logic.

```
// lib/currency.ts (illustrative subset of the 16-currency registry)
export type CurrencyCode = 'USD' | 'EUR' | 'GBP' | 'TND' | 'MAD' | 'DZD' | /* ... */ string;

interface CurrencyMeta {
  code: CurrencyCode;
  locale: string;
  rate: number; // units of this currency per 1 base unit
  exponent: number; // minor-unit digits (e.g. 2 for USD, 3 for TND)
}
```

```

const REGISTRY: Record<string, CurrencyMeta> = {
  USD: { code: 'USD', locale: 'en-US', rate: 1, exponent: 2 },
  EUR: { code: 'EUR', locale: 'fr-FR', rate: 0.92, exponent: 2 },
  TND: { code: 'TND', locale: 'fr-TN', rate: 3.1, exponent: 3 },
  // ... remaining entries
};

const TZ_DEFAULTS: Record<string, CurrencyCode> = {
  'Africa/Tunis': 'TND',
  'Europe/Paris': 'EUR',
  'Europe/London': 'GBP',
  'America/New_York': 'USD',
  // ...
};

export function detectCurrency(): CurrencyCode {
  try {
    const tz = Intl.DateTimeFormat().resolvedOptions().timeZone;
    return TZ_DEFAULTS[tz] ?? 'USD';
  } catch {
    return 'USD';
  }
}

export function formatMoney(baseMinor: number, code: CurrencyCode): string {
  const m = REGISTRY[code] ?? REGISTRY.USD;
  const major = (baseMinor / 100) * m.rate;
  return new Intl.NumberFormat(m.locale, {
    style: 'currency',
    currency: m.code,
    minimumFractionDigits: m.exponent,
    maximumFractionDigits: m.exponent,
  }).format(major);
}

```

Detection precedence: explicit user selection (persisted in [ui](#)) → timezone-derived default → [USD](#) fallback. Because rates are static in this architecture, currency display is a deterministic function and fully testable. Live FX would be the first feature to require an external dependency (see Section 5).

2.7 Trade-offs and Failure Modes

The architecture is deliberately scoped, and its limits are explicit:

- **Storage quota.** [localStorage](#) is typically capped near 5 MB per origin and is synchronous. Persisted slices are kept small via [partialize](#); large or binary assets are never stored.
- **Multi-tab consistency.** Zustand does not synchronize across tabs by default. A [storage](#) event listener that calls [persist.rehydrate\(\)](#) closes this gap:

```

window.addEventListener('storage', (e) => {
  if (e.key === 'sou9:cart') useCartStore.persist.rehydrate();
});

```

- **Trust boundary.** Client state is user-controlled. This architecture is appropriate for demonstration, prototyping, kiosk and offline-first scenarios, or as the front half of a system where an authoritative service validates prices and inventory at payment time. It must not be treated as a source of truth for money movement.
- **No cross-device state.** Persistence is per browser profile. Sync would require a backend, which is an intentional non-goal here.

3. UI/UX: Framer Motion and Custom Tailwind Design Tokens

3.1 Token Architecture

Design decisions are encoded as **tokens** at the Tailwind configuration layer and exposed as CSS custom properties, so components consume semantic names rather than raw values. This yields theme switching without component changes and prevents ad-hoc color or spacing drift.

```

/* app/globals.css */

```

```

:root {
  --color-surface: 255 255 255;
  --color-surface-raised: 248 248 250;
  --color-ink: 17 17 24;
  --color-ink-muted: 98 102 115;
  --color-brand: 14 116 144;
  --color-brand-contrast: 255 255 255;
  --radius-card: 14px;
  --shadow-lift: 0 8px 24px rgb(17 17 24 / 0.08);
  --ease-out-expo: cubic-bezier(0.16, 1, 0.3, 1);
}

.dark {
  --color-surface: 12 12 16;
  --color-surface-raised: 22 22 29;
  --color-ink: 240 240 245;
  --color-ink-muted: 150 154 168;
  --color-brand: 34 190 220;
  --color-brand-contrast: 8 8 12;
  --shadow-lift: 0 8px 24px rgb(0 0 0 / 0.5);
}

// tailwind.config.ts (excerpt)
import type { Config } from 'tailwindcss';

export default {
  darkMode: 'class',
  theme: {
    extend: {
      colors: {
        surface: 'rgb(var(--color-surface) / <alpha-value>)',
        'surface-raised': 'rgb(var(--color-surface-raised) / <alpha-value>)',
        ink: 'rgb(var(--color-ink) / <alpha-value>)',
        'ink-muted': 'rgb(var(--color-ink-muted) / <alpha-value>)',
        brand: 'rgb(var(--color-brand) / <alpha-value>)',
      },
      borderRadius: { card: 'var(--radius-card)' },
      boxShadow: { lift: 'var(--shadow-lift)' },
      transitionTimingFunction: { 'out-expo': 'var(--ease-out-expo)' },
    },
  },
} satisfies Config;

```

Why RGB channel triplets: the `<alpha-value>` placeholder preserves Tailwind's opacity modifiers (`bg-brand/20`) while the underlying value remains a swappable CSS variable. Light and dark themes are a class toggle on the root element, driven by the `ui` store.

3.2 Motion System

Framer Motion is used under a small set of **named motion primitives**, not scattered inline props. This keeps timing, easing and choreography consistent and auditable.

```

// lib/motion.ts
import type { Variants, Transition } from 'framer-motion';

export const spring: Transition = { type: 'spring', stiffness: 380, damping: 32 };

export const fadeUp: Variants = {
  hidden: { opacity: 0, y: 12 },
  show: { opacity: 1, y: 0, transition: { duration: 0.28, ease: [0.16, 1, 0.3, 1] } },
};

export const stagger: Variants = {
  show: { transition: { staggerChildren: 0.04 } },
};

export const drawer: Variants = {
  closed: { x: '100%' },
  open: { x: 0, transition: spring },
};

```

Application patterns:

- **Product grids:** a `stagger` parent with `fadeUp` children produces perceptible hierarchy without blocking interaction.

- **Cart drawer:** `AnimatePresence` handles mount and unmount, with the `drawer` variant driven by the `ui` store's `cartOpen` flag.
- **Line-item reordering and removal:** `layout` and `layoutId` props let siblings animate to new positions through FLIP transforms instead of snapping.
- **Number transitions:** subtotal and badge changes use keyed `motion.span` elements, so updates are felt rather than silently swapped.

```
// components/CartDrawer.tsx (excerpt)
'use client';
import { AnimatePresence, motion } from 'framer-motion';

export function CartDrawer() {
  const open = useUIStore((s) => s.cartOpen);
  const lines = useCartStore((s) => s.lines);

  return (
    <AnimatePresence>
      {open && (
        <motion.aside
          key='cart-drawer'
          variants={drawer}
          initial='closed'
          animate='open'
          exit='closed'
          className='fixed inset-y-0 right-0 w-full max-w-md bg-surface shadow-lift'
        >
          <motion.ul layout>
            <AnimatePresence initial={false}>
              {lines.map((l) => (
                <motion.li
                  key={l.productId}
                  layout
                  initial={{ opacity: 0, height: 0 }}
                  animate={{ opacity: 1, height: 'auto' }}
                  exit={{ opacity: 0, height: 0 }}
                >
                  {/* line item */}
                </motion.li>
              ))}
            </AnimatePresence>
          </motion.ul>
        </motion.aside>
      )}
    </AnimatePresence>
  );
}
```

3.3 Performance and Accessibility Constraints

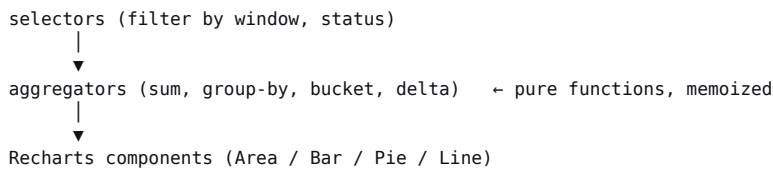
- **Compositor-friendly properties only.** Animations target `transform` and `opacity`. Layout-triggering properties are limited to short, bounded cases such as list item collapse.
- **Reduced motion is honored.** The app wraps the tree in `<MotionConfig reducedMotion='user'>`, which disables transform-based animation for users who request it.
- **Focus management.** The drawer traps focus while open, restores it to the invoking control on close, and closes on `Escape`.
- **Zero-latency feel.** With no network wait, perceived responsiveness is bounded by animation duration. Durations are therefore kept short (roughly 180 to 320 ms) so motion communicates state without becoming the bottleneck.

4. Analytics: Recharts and Dynamic Data Aggregation

4.1 Data Model and Pipeline

The admin console renders operational analytics from two inputs held in the `admin` store: the **30-day daily metrics series** and the **order ledger (17 orders)**. Everything shown is derived on the client:

```
admin store (orders, analytics[30d], windowDays)
  |
  ▼
```



Because every aggregator is a pure function over in-memory arrays of small cardinality, recomputation is effectively free, and charts update in the same frame as the control that changed them.

4.2 Aggregation Primitives

```

// lib/analytics.ts
import type { DailyMetric, Order } from './mockData';

export function sliceWindow(series: DailyMetric[], days: number): DailyMetric[] {
  return series.slice(-days);
}

export function totals(series: DailyMetric[]) {
  return series.reduce(
    (acc, d) => ({
      revenue: acc.revenue + d.revenueBase,
      orders: acc.orders + d.orders,
      visitors: acc.visitors + d.visitors,
    }),
    { revenue: 0, orders: 0, visitors: 0 }
  );
}

// Compare the current window to the immediately preceding window of equal length
export function periodDelta(series: DailyMetric[], days: number) {
  const current = totals(series.slice(-days));
  const previous = totals(series.slice(-days * 2, -days));
  const pct = (a: number, b: number) => (b === 0 ? null : (a - b) / b);
  return {
    revenue: pct(current.revenue, previous.revenue),
    orders: pct(current.orders, previous.orders),
  };
}

export function conversionRate(series: DailyMetric[]): number {
  const t = totals(series);
  return t.visitors === 0 ? 0 : t.orders / t.visitors;
}

export function revenueByCategory(
  orders: Order[],
  productIndex: Map<string, { category: string }>
): { category: string; revenue: number }[] {
  const acc = new Map<string, number>();
  for (const o of orders) {
    if (o.status === 'cancelled') continue;
    for (const l of o.lines) {
      const cat = productIndex.get(l.productId)?.category ?? 'Uncategorized';
      acc.set(cat, (acc.get(cat) ?? 0) + l.quantity * l.unitPriceBase);
    }
  }
  return [...acc].map(([category, revenue]) => ({ category, revenue }));
}

export function statusDistribution(orders: Order[]) {
  const acc: Record<string, number> = {};
  for (const o of orders) acc[o.status] = (acc[o.status] ?? 0) + 1;
  return Object.entries(acc).map(([status, count]) => ({ status, count }));
}

```

Note on the delta calculation: a 30-day seed supports a full 30-day window, or a 15-day window compared against its predecessor. Selecting a window larger than half the series yields `null` deltas, which the UI renders as an em dash rather than a misleading figure.

4.3 Chart Composition

```

// components/admin/RevenueChart.tsx
'use client';
import { useMemo } from 'react';

```

```
import {
  ResponsiveContainer, AreaChart, Area, XAxis, YAxis, Tooltip, CartesianGrid,
} from 'recharts';

export function RevenueChart() {
  const series = useAdminStore((s) => s.analytics);
  const days = useAdminStore((s) => s.windowDays);
  const currency = useUIStore((s) => s.currency);

  const data = useMemo(
    () =>
      sliceWindow(series, days).map((d) => ({
        date: d.date,
        revenue: d.revenueBase,
      })),
    [series, days]
  );

  return (
    <ResponsiveContainer width='100%' height={280}>
      <AreaChart data={data} margin={{ top: 8, right: 8, bottom: 0, left: 0 }}>
        <defs>
          <linearGradient id='rev' x1='0' y1='0' x2='0' y2='1'>
            <stop offset='0%' stopColor='rgb(var(--color-brand))' stopOpacity={0.35} />
            <stop offset='100%' stopColor='rgb(var(--color-brand))' stopOpacity={0} />
          </linearGradient>
        </defs>
        <CartesianGrid vertical={false} strokeOpacity={0.12} />
        <XAxis dataKey='date' tickLine={false} axisLine={false} minTickGap={24} />
        <YAxis
          tickLine={false}
          axisLine={false}
          tickFormatter={(v) => formatMoney(v, currency)}
          width={72}
        />
        <Tooltip formatter={(v: number) => formatMoney(v, currency)} />
        <Area
          dataKey='revenue'
          stroke='rgb(var(--color-brand))'
          fill='url(#rev)'
          strokeWidth={2}
          isAnimationActive
        />
      </AreaChart>
    </ResponsiveContainer>
  );
}
```

Integration points worth noting:

- **Token-aware charting.** Strokes and fills reference the same CSS variables as the rest of the UI, so charts re-theme with the `dark` class without prop changes.
- **Currency-aware axes.** Chart values remain in base minor units; the tooltip and axis formatters convert at render time. Changing currency re-renders labels without re-aggregating data.
- **Stable references.** Selectors return store slices by reference and `useMemo` keys on `[series, days]`, preventing needless chart re-renders.
- **Responsive by construction.** `ResponsiveContainer` binds chart geometry to the layout, so the same component serves desktop and mobile admin views.

4.4 Dashboard Surface

- **KPI cards:** revenue, orders, conversion rate, average order value, each with a `periodDelta` badge.
- **Time-series:** revenue and orders over the selected window (7, 14 or 30 days).
- **Category mix:** a bar or pie of net revenue by category from the order ledger, excluding cancelled orders.
- **Order pipeline:** status distribution across the 17 seeded orders, updating live as admin actions change order status.

Admin mutations (for example, marking an order `shipped`) write to the `admin` store, which persists and immediately re-flows through the aggregators. The dashboard is therefore a **live view over**

mutable local state, not a static report.

5. Conclusion: The Future of Zero-Latency Storefronts

Sou9 demonstrates that a coherent commerce experience (catalog, cart, multi-currency presentation, order management and analytics) can be delivered without a server on the interaction path. The three-store Zustand model, hydration-safe `localStorage` persistence, token-driven UI and pure-function analytics pipeline compose into a system with a small, legible surface area.

What the architecture buys:

- **Latency floor of one frame.** Interactions are bounded by render time, not network time.
- **Operational simplicity.** No database, API tier or scaling plan. Deployment is static asset hosting.
- **Offline tolerance.** After first load, the application continues to function without connectivity.
- **Testability.** Stores, currency logic and aggregators are pure TypeScript and can be unit-tested without mocks of remote services.

What it does not buy, and what comes next:

- **Authoritative state.** Price integrity, inventory reservation and payment require a trusted system. The natural evolution is a *thin authority layer* (payment intents and inventory checks) behind an otherwise client-resident experience.
- **Durable, cross-device persistence.** Replacing `localStorage` with IndexedDB via a custom Zustand `StateStorage` adapter raises capacity and allows asynchronous writes; adding a sync layer (CRDT-based or server-reconciled) enables multi-device continuity.
- **Live reference data.** Static currency rates can be replaced by a periodically refreshed cache, retaining the deterministic formatting path while sourcing rates externally.
- **Edge hybridization.** Because the stores are decoupled from the transport, individual slices (for example, the catalog) can later be hydrated from edge-cached JSON with no change to component code.

The broader trajectory is clear. As browser runtimes, storage APIs and bundlers such as Turbopack mature, the boundary between application and infrastructure moves toward the client. Storefronts that treat the network as an *enhancement* rather than a *prerequisite* will define the zero-latency baseline that users come to expect. Sou9 is a reference implementation of that principle: state local, derivation pure, motion intentional, and the server reserved for the few decisions that genuinely require trust.