CRO Analytics Platform

Technical Architecture & Build Plan

Version 1.0 — February 2026

Shopify-Focused CRO Dashboard SaaS

1. Technology Stack

1.1 Stack Overview

Every technology choice is driven by three constraints: solo AI-assisted developer, Vercel-native deployment, and the need to scale from zero to hundreds of tenants without re-platforming. The stack is entirely TypeScript/JavaScript to minimize context-switching and maximize what AI coding tools can generate effectively.

Layer	Technology	Rationale
Framework	Next.js 14+ (App Router)	Full-stack React framework. API routes, server components, middleware for auth. Deploys natively to Vercel.
Language	TypeScript	Type safety across full stack. Critical for complex data models with multiple API integrations.
Primary DB	PostgreSQL (Supabase)	Multi-tenant relational data, Row Level Security, real-time subscriptions, built-in auth. Free tier generous.
Analytics DB	ClickHouse (ClickHouse Cloud)	Columnar OLAP database for time-series metrics. Handles billions of rows for dashboards. Free tier available.
ORM	Drizzle ORM	Lightweight, type-safe, SQL-first. Better performance than Prisma for complex queries. Native PostgreSQL support.
Auth	Supabase Auth + custom RBAC	OAuth flows for integrations, JWT sessions, row-level security. RBAC layer for workspace/client/store permissions.
Queue / Jobs	Inngest	Serverless event-driven functions. Handles scheduled syncs, retries, fan-out. Free tier: 50K events/mo. Runs on Vercel.
Cache	Vercel KV (Redis)	API response caching, rate limit tracking, session data. Sub-millisecond reads.
File Storage	Vercel Blob / Supabase Storage	Report PDFs, exported CSVs, white-label assets (logos). S3-compatible.
UI Library	shadcn/ui + Tailwind CSS	Copy-paste components (not npm dependency). Full control over styling. Ideal for dashboard UIs.
Charts	Recharts + Tremor	Recharts for custom visualizations. Tremor for pre-built dashboard components (KPI cards, sparklines).
Hosting	Vercel (Pro plan)	Native Next.js deployment, edge functions, cron jobs, serverless functions. Pro plan ($20/mo) for commercial use.
Monitoring	Sentry + Vercel Analytics	Error tracking, performance monitoring, web vitals. Sentry free tier sufficient for MVP.

1.2 Why This Stack (Trade-offs)

Next.js on Vercel vs. Alternatives

Why not a separate backend (Express, Fastify, NestJS)? A standalone API server gives you more control, but it means managing two deployments, two CI pipelines, CORS configuration, and separate scaling. Next.js API routes and Server Actions give you a full backend within the same project. For a solo developer, this is the right trade-off. You can always extract services later if a specific endpoint needs dedicated scaling.

Why not Remix or SvelteKit? Both are excellent, but Next.js has the largest ecosystem of AI-generated code examples, the deepest Vercel integration, and the most community resources for solving edge cases. When you hit a wall at 2am and need to search for a solution, Next.js will have 10x more answers.

Supabase vs. PlanetScale vs. Neon

Supabase wins for this project because it bundles PostgreSQL, authentication, Row Level Security, real-time subscriptions, and file storage into one platform. PlanetScale (MySQL) and Neon (PostgreSQL) are excellent databases, but they are just databases. You would need to add separate services for auth (Clerk, Auth.js), file storage (S3), and real-time features. Supabase gives you all of these with a single account, a single dashboard, and a generous free tier.

The catch: Supabase adds a dependency. If Supabase has an outage, your auth and database both go down. For an MVP, this is an acceptable risk. If the product succeeds, you can migrate PostgreSQL to a standalone instance and swap auth providers without rewriting your app logic, since Drizzle ORM abstracts the database layer.

Inngest vs. BullMQ vs. Vercel Cron

Inngest is purpose-built for serverless. BullMQ requires a persistent Redis connection and a long-running worker process, which means running a separate server outside Vercel. Vercel Cron can trigger functions on a schedule, but it has no retry logic, no fan-out, no event chaining, and a 60-second execution limit on Hobby plans. Inngest runs as serverless functions on Vercel, handles retries automatically, supports complex workflows (sync Store A, then Store B, then aggregate), and has a generous free tier of 50,000 events per month.

1.3 The Dual Database Architecture

This is the most important architectural decision in the entire project. Your app has two fundamentally different data access patterns, and trying to serve both from a single PostgreSQL database will become a bottleneck as you scale.

Concern	PostgreSQL (Supabase)	ClickHouse
Purpose	Application state, user data, configs, integration credentials, dashboard definitions	Time-series metrics, aggregated analytics data, historical trend data for widgets
Query Pattern	OLTP: Small reads/writes, joins, transactions. "Get this user\'s dashboards."	OLAP: Large scans, aggregations, time ranges. "Sessions by device last 30 days."
Scale	Thousands of rows per tenant. Grows with users and configs.	Millions to billions of metric rows. Grows with data sync frequency.
V1 Reality	Primary database from day one. All app logic lives here.	Can defer to V2. Use PostgreSQL with materialized views initially, migrate hot metrics to ClickHouse when needed.

V1 Strategy: Start with PostgreSQL only. Store synced metrics in well-indexed tables with time-based partitioning. Use materialized views for dashboard queries. This will handle your first 20-50 stores comfortably. When dashboard queries start taking more than 2-3 seconds, that is your signal to introduce ClickHouse for the metrics layer. The key is designing your data model so this migration is a backend change, not a frontend rewrite. Your dashboard widgets should query a metrics service abstraction, not raw SQL.

2. Database Schema Design

2.1 Multi-Tenancy Model

The schema follows a strict hierarchy: Organization (top-level billing entity) → Workspace (logical grouping, maps to an agency or brand) → Store (individual Shopify store connection) → Integration (connected tool per store) → Metric Data (synced data points). Every table below the organization level includes tenant isolation columns that enable Row Level Security.

Core Entity Relationships

```
Organization (1) → (*) Workspace (1) → (*) Store (1) → (*) Integration
Organization (1) → (*) User  (via org_members with role)
Workspace    (1) → (*) User  (via workspace_members with role)
Workspace    (1) → (*) Dashboard (1) → (*) Widget
Store        (1) → (*) MetricData (partitioned by time + source)
```

Key Tables

organizations — Top-level billing and account entity

```
id (uuid PK), name, slug (unique), plan (enum), billing_email,
stripe_customer_id, white_label_config (jsonb), created_at, updated_at
```

The white_label_config JSONB column stores branding overrides: logo URL, primary color, custom domain, favicon, email sender name. Stored as JSONB so the schema is flexible without migrations. This is the "flip the switch" field for white labeling.

workspaces — Logical grouping (agency = one workspace per client, brand = one workspace)

```
id (uuid PK), org_id (FK), name, slug, settings (jsonb), created_at
```

stores — Individual Shopify store connections

```
id (uuid PK), workspace_id (FK), shopify_domain, shopify_access_token
(encrypted),
shopify_store_name, currency, timezone, is_active, last_synced_at, created_at
```

integrations — Connected third-party tools per store

```
id (uuid PK), store_id (FK), provider (enum: clarity|ga4|intelligems|
  knocommerce|gorgias|judgeme), credentials (jsonb, encrypted),
config (jsonb), status (enum: active|error|disconnected),
last_synced_at, last_error, sync_frequency (interval), created_at
```

The credentials JSONB stores provider-specific auth data: API keys, OAuth tokens, refresh tokens, project IDs. Encrypted at rest using Supabase Vault or application-level encryption. The config JSONB stores per-integration settings like which Clarity dimensions to sync or which Gorgias tags to track.

users + org_members + workspace_members — RBAC hierarchy

```
users: id, email, name, avatar_url, auth_provider, created_at
org_members: user_id, org_id, role (owner|admin|member|viewer)
workspace_members: user_id, workspace_id, role (admin|editor|viewer)
```

Users can belong to multiple organizations (a freelancer working with multiple agencies). Org-level roles control billing and org settings. Workspace-level roles control data access. Client portal users will be "viewer" role members of a specific workspace, seeing only dashboards shared with their workspace.

2.2 Dashboard & Widget Schema

dashboards — User-created or template dashboards

```
id (uuid PK), workspace_id (FK), created_by (FK users),
name, description, layout (jsonb), is_template (boolean),
is_preset (boolean), visibility (enum: private|workspace|public),
store_filter (uuid[] | null), created_at, updated_at
```

The layout JSONB stores the column configuration and widget ordering. A simplified structure that supports your 1/2/3 column layout:

```
{
  "columns": 2,
  "sections": [
    {
      "id": "sec_1", "title": "Traffic Overview",
      "widgets": ["wid_abc", "wid_def", "wid_ghi"]
    },
    {
      "id": "sec_2", "title": "A/B Test Results",
      "widgets": ["wid_jkl", "wid_mno"]
    }
  ]
}
```

widgets — Individual data visualization components

```
id (uuid PK), dashboard_id (FK), type (enum: kpi_card|line_chart|
  bar_chart|table|funnel|pie_chart|heatmap_summary|review_feed|
  sentiment_gauge|experiment_card|survey_summary),
title, data_source (enum: shopify|ga4|clarity|intelligems|
  knocommerce|gorgias|judgeme|computed),
query_config (jsonb), display_config (jsonb),
position (integer), size (enum: small|medium|large|full),
created_at, updated_at
```

The query_config JSONB defines what data the widget fetches. This is the critical abstraction layer. The frontend never constructs raw SQL. Instead, it sends a declarative config that the backend metrics service interprets:

```
{
  "metric": "sessions",
  "source": "ga4",
  "aggregation": "sum",
  "time_range": "last_30_days",
  "group_by": "device_category",
  "filters": { "country": "US" },
  "compare_to": "previous_period"
}
```

The display_config JSONB controls visual presentation: colors, number formatting, chart axis labels, conditional formatting rules. Separating query from display means the same data can power a KPI card in one dashboard and a line chart in another.

2.3 Metrics Data Layer (AI-Ready)

This is the schema that makes or breaks the AI layer. The goal is a normalized metrics store where every data point, regardless of source, follows the same structure. This means an AI agent can query across all data sources using a single, predictable schema rather than needing to understand 7 different API response formats.

metric_events — The unified metrics table

```
id (bigint PK, auto-increment),
store_id (uuid FK, NOT NULL),
source (enum: shopify|ga4|clarity|intelligems|knocommerce|gorgias|judgeme),
metric_name (text, NOT NULL),  -- e.g., 'sessions', 'scroll_depth',
'csat_score'
metric_value (numeric, NOT NULL),
dimensions (jsonb),  -- e.g., {"device": "mobile", "country": "US", "page":
"/products"}
recorded_at (timestamptz, NOT NULL),  -- when the metric occurred
synced_at (timestamptz, NOT NULL),  -- when we ingested it
raw_data (jsonb),  -- original API response (for debugging and re-processing)
metadata (jsonb)  -- source-specific context
```

Partition this table by (store_id, recorded_at) for efficient time-range queries scoped to a single store. Index on (store_id, source, metric_name, recorded_at) for the most common dashboard query pattern.

Why this design is AI-ready:

- Every metric follows the same structure: what happened (metric_name), how much (metric_value), when (recorded_at), and context (dimensions).
- An AI agent can be told: "Query metric_events where source = 'clarity' and metric_name = 'scroll_depth' for the last 7 days grouped by dimensions->device" without needing to know how Clarity\'s API works.
- The raw_data column preserves the original API response, so you can always re-process data if your normalization logic improves.
- The dimensions JSONB is flexible enough to store source-specific breakdowns without schema changes.

metric_definitions — Registry of all known metrics

```
id (text PK),  -- e.g., 'clarity.scroll_depth'
source, metric_name, display_name, description, unit (enum: count|
  percentage|currency|duration|score|rating), aggregation_default
  (enum: sum|avg|min|max|count|latest), category (enum: traffic|
  conversion|engagement|sentiment|testing|revenue)
```

This table serves double duty: it drives the widget picker UI (users browse available metrics by category) and it tells the AI layer what each metric means, how to aggregate it, and what unit to display.

2.4 Sync State & Job Tracking

sync_jobs — Track every data sync operation

```
id (uuid PK), integration_id (FK), status (enum: pending|running|
  completed|failed|cancelled), started_at, completed_at,
records_synced (integer), error_message, error_details (jsonb),
sync_type (enum: full|incremental|webhook), cursor (jsonb)
```

The cursor JSONB stores pagination state for APIs that require incremental syncing. For example, Shopify order sync would store the last order ID processed so the next sync picks up where it left off. This is critical for reliability: if a sync fails midway, the next attempt resumes from the cursor rather than re-fetching everything.

2.5 Row Level Security Strategy

Supabase RLS policies ensure data isolation at the database level. Even if application code has a bug, one organization can never see another organization\'s data.

- Every query automatically filters by the authenticated user\'s org_id
- Workspace-level access checks happen in RLS policies, not application code
- Service role (for sync jobs) bypasses RLS, but only runs in server-side functions
- Client-side Supabase calls always go through RLS

```
-- Example RLS policy for stores table
CREATE POLICY "Users can view stores in their workspaces"
  ON stores FOR SELECT USING (
    workspace_id IN (
      SELECT workspace_id FROM workspace_members
      WHERE user_id = auth.uid()
    )
  );
```

3. Integration Architecture

3.1 Integration Registry

Each third-party tool connects through a standardized integration interface. This is the full API compatibility matrix based on research:

Integration	API Type	Auth Method	Data Available	Sync Strategy
Shopify	REST + GraphQL	OAuth 2.0 (App install)	Orders, products, customers, carts, checkouts	Webhooks + daily full sync
GA4	REST (Data API)	OAuth 2.0 (Google)	Sessions, conversions, page views, events, funnels	Scheduled pull (hourly)
Microsoft Clarity	REST (Export API)	Bearer token	Engagement time, scroll depth, dead clicks, rage clicks, device breakdown	Scheduled pull (daily)
Intelligems	Webhooks + JS API	Webhook signature	Experiment events, test groups, statistical significance, order attribution	Inbound webhooks (real-time)
KnoCommerce	REST API	API Key	Post-purchase survey responses, attribution data	Scheduled pull (daily)
Gorgias	REST API	API Key + Domain	Tickets, satisfaction scores, tags, response times	Webhooks + scheduled pull
Judge.me	REST API	API Key	Reviews, ratings, pending reviews, product-level aggregates	Scheduled pull (daily)

3.2 Data Flow Pipeline

All data flows through the same pipeline regardless of source. This is the unified process:

1. Trigger: Inngest cron fires (hourly/daily per schedule) OR webhook received from Intelligems/Gorgias/Shopify
2. Fetch: Integration adapter calls the third-party API using stored credentials. Handles pagination, rate limits, retries.

3. Transform: Raw API response is normalized into the metric_events schema. Source-specific fields go into dimensions and raw_data JSONB columns.
4. Load: Normalized records are batch-inserted into metric_events. Sync cursor is updated in sync_jobs.
5. Notify: Dashboard cache is invalidated for affected store. If real-time subscriptions are active, Supabase broadcast triggers widget refresh.

Integration Adapter Pattern

Each integration implements a common interface. This makes adding new integrations predictable:

```
interface IntegrationAdapter {
  connect(credentials: unknown): Promise<ConnectionResult>;
  disconnect(integrationId: string): Promise<void>;
  testConnection(integrationId: string): Promise<boolean>;
  sync(integration: Integration, cursor?: SyncCursor): Promise<SyncResult>;
  getAvailableMetrics(): MetricDefinition[];
}
```

Every adapter (ClarityAdapter, GA4Adapter, GorgiasAdapter, etc.) implements this interface. The sync engine doesn\'t know or care which API it\'s talking to. It just calls adapter.sync() and gets back normalized metric_events.

Webhook Ingestion

For Intelligems, Gorgias, and Shopify, you receive real-time webhooks. These hit a Next.js API route at /api/webhooks/[provider], which validates the signature, creates an Inngest event, and returns 200 immediately. Inngest then processes the payload asynchronously. This ensures you never lose webhook data due to processing timeouts.

3.3 Credential Security

Third-party API credentials are the most sensitive data in the system. The security approach:

- OAuth tokens (Shopify, GA4): Stored in Supabase with column-level encryption via Supabase Vault. Refresh tokens are rotated automatically.
- API keys (Clarity, Gorgias, KnoCommerce, Judge.me): Encrypted at the application level before storage. Decrypted only in server-side functions, never exposed to the client.
- Webhook secrets (Intelligems): Stored as environment variables in Vercel, not in the database.
- Access pattern: Frontend never sees raw credentials. All API calls to third parties happen in server-side functions or Inngest jobs.

4. Infrastructure Costs

4.1 Cost Breakdown by Phase

Service	Free Tier	Paid Tier	When to Upgrade
Vercel (Hosting)	Hobby (non-commercial)	Pro: $20/mo	Immediately (commercial use)
Supabase (PostgreSQL + Auth)	500MB DB, 50K MAU	Pro: $25/mo	~50 active stores or auth limits
ClickHouse Cloud	Trial credits	~$50-100/mo	V2: when metric volume exceeds PG
Inngest (Queue)	50K events/mo	$50/mo	~20 stores syncing hourly
Vercel KV (Redis)	3K requests/day	$7/mo (100K/day)	When caching is needed
Sentry (Monitoring)	5K errors/mo	$26/mo	Production launch
Domain + Email	N/A	~$20/yr + $5/mo	Immediately

MVP Phase (0-20 stores): ~$45/month. Vercel Pro ($20) + Supabase free tier + Inngest free tier + domain/email ($7). This is your burn rate during development and beta.

Growth Phase (20-100 stores): ~$120-200/month. Supabase Pro ($25) + Inngest paid ($50) + Vercel Pro ($20) + monitoring + KV. At this point you should have paying customers covering these costs.

Scale Phase (100+ stores): ~$300-500/month. Add ClickHouse Cloud for metrics, higher Supabase tier, more Inngest events. Revenue should be $5K-15K/month at this stage.

5. Phased Build Plan

5.1 Phase 1: Foundation (Weeks 1-4)

Goal: Working multi-tenant app shell with auth, one integration (Shopify), and basic data display.

Week 1-2: Project scaffold and auth

- Next.js project with TypeScript, Tailwind, shadcn/ui
- Supabase project setup: database, auth, RLS policies
- Drizzle ORM schema for organizations, workspaces, users, org_members, workspace_members
- Auth flows: sign up, sign in, invite member, org switching
- Basic layout: sidebar navigation, workspace selector, settings pages

Week 3-4: Shopify integration + data model

- Shopify OAuth app installation flow
- Store connection: save credentials, verify access, pull store metadata
- Integration adapter for Shopify: sync orders, products, basic analytics
- metric_events table + first data sync via Inngest
- Simple data display: table of recent orders, basic KPI cards (revenue, order count, AOV)

5.2 Phase 2: Dashboard Engine (Weeks 5-8)

Goal: Widget system, dashboard builder, 2-3 more integrations.

Week 5-6: Widget framework + dashboard builder

- Widget component system: KPI card, line chart, bar chart, table, funnel
- Dashboard creation: name, choose columns (1/2/3), add widgets from picker
- Widget configuration panel: select data source, metric, time range, display options
- Dashboard save/load, widget reordering within sections
- Pre-built dashboard templates (admin-created, stored with is_template flag)

Week 7-8: GA4 + Clarity integrations

- GA4 OAuth flow + Data API adapter: sessions, conversions, page views, device breakdown
- Clarity API adapter: engagement metrics, scroll depth, dead clicks, device breakdown

- Normalize both into metric_events. Widgets can now display multi-source data.
- Dashboard templates: "CRO Overview" combining Shopify revenue + GA4 traffic + Clarity engagement

5.3 Phase 3: CRO-Specific Features (Weeks 9-12)

Goal: Intelligems, Gorgias, reviews, and the features that make this a CRO tool, not just another analytics dashboard.

Week 9-10: A/B testing + customer voice

- Intelligems webhook receiver + experiment card widget (test name, groups, confidence, lift)
- KnoCommerce adapter: survey responses, attribution breakdown
- Survey summary widget and sentiment analysis display

Week 11-12: Support + reviews + polish

- Gorgias adapter: ticket volume, CSAT, negative ticket feed, tag-based filtering
- Judge.me adapter: review feed, rating distribution, pending review alerts
- Sentiment gauge widget pulling from Gorgias + Judge.me data
- Dashboard sharing: generate read-only link for a dashboard (precursor to client portal)
- Beta testing with 10-20 users from your network

5.4 Phase 4: Post-Beta Iteration (Weeks 13-16)

Goal: Respond to beta feedback, add reporting, begin monetization.

- PDF/email report generation from dashboards
- Scheduled report delivery (weekly/monthly via Inngest)
- Client portal: separate login for workspace viewers with limited navigation
- Billing integration (Stripe) based on store count
- Performance optimization: caching layer, query optimization, lazy-loading widgets

5.5 Future Phases (Post-Launch)

- AI analysis layer: natural language queries across metric_events ("Why did conversion drop last week?")
- White labeling UI: custom branding, custom domains, branded emails
- ClickHouse migration for metrics layer when PostgreSQL becomes the bottleneck
- Automated CRO insights: anomaly detection, trend alerts, opportunity scoring

- Additional integrations: Hotjar, Lucky Orange, Rebuy, Klaviyo post-purchase
- Shopify App Store listing for distribution

6. Project Structure

Recommended Next.js App Router project structure:

```
/app
  /(auth)                -- Login, signup, invite accept
  /(dashboard)           -- Main authenticated app
    /[orgSlug]           -- Org-scoped routes
      /[workspaceSlug] -- Workspace-scoped routes
        /dashboards      -- Dashboard list + builder
        /stores          -- Store management + connections
        /integrations    -- Integration setup + status
        /settings        -- Workspace settings
      /settings          -- Org settings, billing, members
  /api
    /webhooks/[provider]  -- Webhook receivers
    /integrations/[id]    -- Integration CRUD
    /metrics              -- Metrics query endpoint
    /sync                 -- Manual sync triggers
/components
  /widgets               -- Widget components (KPI, chart, etc.)
  /dashboard             -- Dashboard builder components
  /integrations          -- Integration setup UI
  /ui                    -- shadcn/ui primitives
/lib
  /db                    -- Drizzle schema + queries
  /integrations          -- Integration adapters
    /shopify             -- Shopify adapter
    /ga4                 -- GA4 adapter
    /clarity             -- Clarity adapter
    /intelligems         -- Intelligems adapter
    /gorgias             -- Gorgias adapter
    /knocommerce         -- KnoCommerce adapter
    /judgeme             -- Judge.me adapter
  /metrics               -- Metrics service (query builder)
  /inngest               -- Inngest functions (sync jobs)
  /auth                  -- Auth utilities + RBAC helpers
```

6.1 Key Architectural Principles

- **Adapter pattern for integrations:** Every third-party tool gets an adapter that implements the same interface. Adding a new integration means writing one file, not touching the dashboard or sync engine.
- **Metrics service abstraction:** Widgets never query the database directly. They call a metrics service that translates widget query_config into SQL, handles caching, and returns formatted results. This is the layer that swaps from PostgreSQL to ClickHouse without frontend changes.
- **Server Components by default:** Dashboard pages use React Server Components for initial data fetch. Widget data refreshes via client-side SWR/React Query with stale-while-revalidate caching.
- **Feature flags from day one:** Use a simple JSONB column on organizations for feature flags. This lets you gate features by plan tier, run beta tests, and gradually roll out new integrations.