SAINTRIX Beta Technical Architecture & Database Schema

1. Introduction

This document provides a comprehensive technical architecture and database schema design for SAINTRIX Beta, a credit repair software platform. The architecture leverages modern cloud-native technologies including Supabase for backend services, Vercel for frontend deployment, and Next.js for the application framework. This design prioritizes security, scalability, and maintainability while ensuring compliance with financial data protection standards.

The architecture follows a serverless-first approach, utilizing Supabase's Backend-as-a-Service (BaaS) capabilities to minimize infrastructure management overhead while maximizing development velocity. The system is designed to handle sensitive financial data with enterprise-grade security measures, including end-to-end encryption, row-level security, and comprehensive audit logging.

2. System Architecture Overview

2.1. High-Level Architecture

The SAINTRIX Beta system follows a three-tier architecture pattern with clear separation of concerns between presentation, application logic, and data layers. The architecture is designed to be cloud-native, leveraging serverless technologies for automatic scaling and reduced operational overhead.

```
graph TB
  subgraph "Client Layer"
    A[Client Browser - React/Next.js]
    B[Admin Browser - React/Next.js]
  end

  subgraph "Vercel Edge Network"
    C[Vercel CDN]
    D[Next.js API Routes]
    E[Static Assets]
  end
```

```
    subgraph "Supabase Backend"
        F[Supabase API Gateway]
        G[PostgreSQL Database]
        H[Supabase Auth]
        I[Supabase Storage]
        J[Edge Functions Runtime]
    end

    subgraph "External Services"
        K[Credit Data Provider API]
        L[Email Service Provider]
        M[PDF Generation Service]
    end

    A --> C
    B --> C
    C --> D
    D --> F
    F --> G
    F --> H
    F --> I
    F --> J
    J --> K
    J --> L
    J --> M

    style A fill:#e1f5fe
    style B fill:#e8f5e8
    style G fill:#fff3e0
    style H fill:#fce4ec
    style I fill:#f3e5f5
```

2.2. Technology Stack

Frontend Layer:
- **Framework:** Next.js 14+ with React 18+
- **Styling:** Tailwind CSS for utility-first styling
- **State Management:** React Context API with useReducer for complex state
- **Form Handling:** React Hook Form with Zod validation
- **UI Components:** Headless UI or Radix UI for accessible components
- **Authentication:** Supabase Auth SDK
- **HTTP Client:** Supabase JavaScript client

Backend Layer:
- **Database:** Supabase PostgreSQL with Row-Level Security (RLS)
- **Authentication:** Supabase Auth with JWT tokens
- **File Storage:** Supabase Storage with encryption at rest

- **Serverless Functions:** Supabase Edge Functions (Deno runtime)
- **Real-time:** Supabase Realtime for live updates

Deployment & Infrastructure:
- **Frontend Hosting:** Vercel with global CDN
- **Backend Services:** Supabase cloud infrastructure
- **Domain & SSL:** Vercel custom domains with automatic SSL
- **Monitoring:** Vercel Analytics and Supabase Dashboard

Development Tools:
- **IDE:** Cursor AI for AI-assisted development
- **Version Control:** Git with GitHub integration
- **CI/CD:** Vercel automatic deployments
- **Testing:** Jest for unit tests, Playwright for E2E testing

2.3. Security Architecture

Security is paramount in SAINTRIX Beta due to the sensitive nature of financial and personal data. The security architecture implements defense-in-depth principles with multiple layers of protection.

Authentication & Authorization:
- Multi-factor authentication (MFA) support through Supabase Auth
- JWT-based session management with automatic token refresh
- Role-based access control (RBAC) with granular permissions
- Row-level security (RLS) policies enforced at the database level

Data Protection:
- Encryption at rest for all database data and file storage
- TLS 1.3 encryption for all data in transit
- Field-level encryption for highly sensitive data (SSN, account numbers)
- Secure key management through Supabase Vault

Application Security:
- Input validation and sanitization on all user inputs
- SQL injection prevention through parameterized queries
- Cross-site scripting (XSS) protection via Content Security Policy
- Cross-site request forgery (CSRF) protection
- Rate limiting on all API endpoints

Compliance & Auditing:
- Comprehensive audit logging for all data access and modifications
- Data retention policies aligned with regulatory requirements

- GDPR compliance features including data export and deletion
- SOC 2 Type II compliance through Supabase infrastructure

3. Database Schema Design

3.1. Core Entity Relationship Model

The database schema is designed to efficiently store and manage credit repair data while maintaining referential integrity and supporting complex queries. The schema follows normalization principles to minimize data redundancy while optimizing for read performance.

```
erDiagram
  auth_users ||--|| profiles : "has"
  profiles ||--o{ credit_reports : "owns"
  credit_reports ||--o{ credit_accounts : "contains"
  credit_reports ||--o{ credit_inquiries : "contains"
  credit_reports ||--o{ public_records : "contains"
  credit_accounts ||--o{ dispute_items : "disputed_as"
  credit_inquiries ||--o{ dispute_items : "disputed_as"
  public_records ||--o{ dispute_items : "disputed_as"
  profiles ||--o{ dispute_items : "creates"
  dispute_items ||--o{ dispute_letters : "generates"
  dispute_items ||--o{ documents : "supports"
  profiles ||--o{ documents : "uploads"
  profiles ||--o{ admin_tasks : "assigned_to"
  dispute_items ||--o{ admin_tasks : "relates_to"
  profiles ||--o{ audit_logs : "performed_by"
  profiles ||--o{ client_messages : "receives"
  profiles ||--o{ letter_templates : "created_by"
```

3.2. Detailed Table Specifications

3.2.1. User Management Tables

profiles

```
CREATE TABLE profiles (
  id UUID PRIMARY KEY REFERENCES auth.users(id) ON DELETE CASCADE,
  email TEXT UNIQUE NOT NULL,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL,
  phone_number TEXT,
  date_of_birth DATE,
  ssn_encrypted TEXT, -- Encrypted SSN for identity verification
```

```
    address_line1 TEXT,
    address_line2 TEXT,
    city TEXT,
    state TEXT,
    zip_code TEXT,
    user_type TEXT NOT NULL DEFAULT 'client' CHECK (user_type IN ('client',
'admin', 'super_admin')),
    client_status TEXT DEFAULT 'active' CHECK (client_status IN ('active', 'on_hold',
'completed', 'suspended')),
    onboarding_completed BOOLEAN DEFAULT FALSE,
    terms_accepted_at TIMESTAMP WITH TIME ZONE,
    privacy_policy_accepted_at TIMESTAMP WITH TIME ZONE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    last_login_at TIMESTAMP WITH TIME ZONE
);

-- Row Level Security Policies
ALTER TABLE profiles ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Users can view own profile" ON profiles
    FOR SELECT USING (auth.uid() = id);

CREATE POLICY "Users can update own profile" ON profiles
    FOR UPDATE USING (auth.uid() = id);

CREATE POLICY "Admins can view all profiles" ON profiles
    FOR SELECT USING (
        EXISTS (
            SELECT 1 FROM profiles
            WHERE id = auth.uid()
            AND user_type IN ('admin', 'super_admin')
        )
    );
```

3.2.2. Credit Data Tables

credit_reports

```
 CREATE TABLE credit_reports (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID NOT NULL REFERENCES profiles(id) ON DELETE CASCADE,
    bureau_name TEXT NOT NULL CHECK (bureau_name IN ('equifax', 'experian',
'transunion')),
    report_date DATE NOT NULL,
    credit_score INTEGER,
    score_model TEXT, -- FICO, VantageScore, etc.
    raw_data JSONB, -- Store complete raw API response
    parsed_data JSONB, -- Store structured parsed data
    import_status TEXT DEFAULT 'pending' CHECK (import_status IN ('pending',
```

```
'processing', 'completed', 'failed')),
    import_error_message TEXT,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Indexes for performance
CREATE INDEX idx_credit_reports_user_id ON credit_reports(user_id);
CREATE INDEX idx_credit_reports_bureau_date ON credit_reports(bureau_name,
report_date);

-- RLS Policies
ALTER TABLE credit_reports ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Users can view own credit reports" ON credit_reports
    FOR SELECT USING (
        user_id = auth.uid() OR
        EXISTS (
            SELECT 1 FROM profiles
            WHERE id = auth.uid()
            AND user_type IN ('admin', 'super_admin')
        )
    );
```

credit_accounts

```
 CREATE TABLE credit_accounts (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    credit_report_id UUID NOT NULL REFERENCES credit_reports(id) ON DELETE
CASCADE,
    account_number_encrypted TEXT, -- Encrypted account number
    account_name TEXT NOT NULL,
    account_type TEXT NOT NULL CHECK (account_type IN ('credit_card',
'mortgage', 'auto_loan', 'personal_loan', 'student_loan', 'other')),
    creditor_name TEXT NOT NULL,
    account_status TEXT CHECK (account_status IN ('open', 'closed', 'paid',
'charged_off', 'collection')),
    balance DECIMAL(12,2),
    credit_limit DECIMAL(12,2),
    payment_history JSONB, -- Array of payment statuses by month
    date_opened DATE,
    date_closed DATE,
    last_payment_date DATE,
    is_negative BOOLEAN DEFAULT FALSE,
    negative_reason TEXT, -- late_payment, charge_off, collection, etc.
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE INDEX idx_credit_accounts_report_id ON credit_accounts(credit_report_id);
CREATE INDEX idx_credit_accounts_negative ON credit_accounts(is_negative)
```

```
WHERE is_negative = TRUE;

ALTER TABLE credit_accounts ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Users can view own credit accounts" ON credit_accounts
    FOR SELECT USING (
        EXISTS (
            SELECT 1 FROM credit_reports cr
            WHERE cr.id = credit_report_id
            AND (cr.user_id = auth.uid() OR
                EXISTS (SELECT 1 FROM profiles WHERE id = auth.uid() AND user_type
IN ('admin', 'super_admin')))
        )
    );
```

credit_inquiries

```
 CREATE TABLE credit_inquiries (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    credit_report_id UUID NOT NULL REFERENCES credit_reports(id) ON DELETE
CASCADE,
    inquiry_type TEXT NOT NULL CHECK (inquiry_type IN ('hard', 'soft')),
    creditor_name TEXT NOT NULL,
    inquiry_date DATE NOT NULL,
    purpose TEXT, -- auto, mortgage, credit_card, etc.
    is_disputed BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE INDEX idx_credit_inquiries_report_id ON credit_inquiries(credit_report_id);
CREATE INDEX idx_credit_inquiries_date ON credit_inquiries(inquiry_date);

ALTER TABLE credit_inquiries ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Users can view own credit inquiries" ON credit_inquiries
    FOR SELECT USING (
        EXISTS (
            SELECT 1 FROM credit_reports cr
            WHERE cr.id = credit_report_id
            AND (cr.user_id = auth.uid() OR
                EXISTS (SELECT 1 FROM profiles WHERE id = auth.uid() AND user_type
IN ('admin', 'super_admin')))
        )
    );
```

3.2.3. Dispute Management Tables

dispute_items

```sql
CREATE TABLE dispute_items (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID NOT NULL REFERENCES profiles(id) ON DELETE CASCADE,
    credit_account_id UUID REFERENCES credit_accounts(id) ON DELETE SET NULL,
    credit_inquiry_id UUID REFERENCES credit_inquiries(id) ON DELETE SET NULL,
    public_record_id UUID REFERENCES public_records(id) ON DELETE SET NULL,
    dispute_type TEXT NOT NULL CHECK (dispute_type IN ('account', 'inquiry',
'public_record', 'personal_info')),
    dispute_reason TEXT NOT NULL, -- not_mine, paid_in_full, incorrect_balance, etc.
    client_notes TEXT,
    admin_notes TEXT,
    status TEXT DEFAULT 'pending_review' CHECK (status IN (
        'pending_review', 'approved', 'rejected', 'letter_generated',
        'letter_sent', 'response_received', 'resolved', 'unresolved'
    )),
    priority TEXT DEFAULT 'medium' CHECK (priority IN ('low', 'medium', 'high',
'urgent')),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    resolved_at TIMESTAMP WITH TIME ZONE,

    -- Ensure only one reference is set
    CONSTRAINT dispute_item_single_reference CHECK (
        (credit_account_id IS NOT NULL)::int +
        (credit_inquiry_id IS NOT NULL)::int +
        (public_record_id IS NOT NULL)::int = 1
    )
);

CREATE INDEX idx_dispute_items_user_id ON dispute_items(user_id);
CREATE INDEX idx_dispute_items_status ON dispute_items(status);
CREATE INDEX idx_dispute_items_created_at ON dispute_items(created_at);

ALTER TABLE dispute_items ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Users can view own dispute items" ON dispute_items
    FOR SELECT USING (
        user_id = auth.uid() OR
        EXISTS (
            SELECT 1 FROM profiles
            WHERE id = auth.uid()
            AND user_type IN ('admin', 'super_admin')
        )
    );

CREATE POLICY "Users can create own dispute items" ON dispute_items
    FOR INSERT WITH CHECK (user_id = auth.uid());

CREATE POLICY "Admins can update dispute items" ON dispute_items
    FOR UPDATE USING (
        EXISTS (
```

```
        SELECT 1 FROM profiles
        WHERE id = auth.uid()
        AND user_type IN ('admin', 'super_admin')
      )
  );
```

dispute_letters

```
 CREATE TABLE dispute_letters (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  dispute_item_id UUID NOT NULL REFERENCES dispute_items(id) ON DELETE
CASCADE,
  template_id UUID REFERENCES letter_templates(id),
  letter_content TEXT NOT NULL,
  letter_pdf_path TEXT, -- Path to generated PDF in Supabase Storage
  status TEXT DEFAULT 'draft' CHECK (status IN ('draft', 'generated', 'sent',
'response_received')),
  sent_date DATE,
  response_deadline DATE,
  bureau_response TEXT,
  response_received_date DATE,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
  updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE INDEX idx_dispute_letters_dispute_item ON
dispute_letters(dispute_item_id);
CREATE INDEX idx_dispute_letters_status ON dispute_letters(status);

ALTER TABLE dispute_letters ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Users can view own dispute letters" ON dispute_letters
  FOR SELECT USING (
    EXISTS (
      SELECT 1 FROM dispute_items di
      WHERE di.id = dispute_item_id
      AND (di.user_id = auth.uid() OR
          EXISTS (SELECT 1 FROM profiles WHERE id = auth.uid() AND user_type
IN ('admin', 'super_admin')))
    )
  );
```

3.2.4. Document Management Tables

documents

```
 CREATE TABLE documents (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id UUID NOT NULL REFERENCES profiles(id) ON DELETE CASCADE,
```

```
    dispute_item_id UUID REFERENCES dispute_items(id) ON DELETE SET NULL,
    file_name TEXT NOT NULL,
    file_path TEXT NOT NULL, -- Path in Supabase Storage
    file_size BIGINT NOT NULL,
    file_type TEXT NOT NULL,
    document_type TEXT CHECK (document_type IN ('evidence', 'identity',
'bureau_response', 'internal')),
    description TEXT,
    uploaded_by UUID REFERENCES profiles(id),
    is_client_visible BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE INDEX idx_documents_user_id ON documents(user_id);
CREATE INDEX idx_documents_dispute_item ON documents(dispute_item_id);

ALTER TABLE documents ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Users can view own documents" ON documents
    FOR SELECT USING (
        (user_id = auth.uid() AND is_client_visible = TRUE) OR
        EXISTS (
            SELECT 1 FROM profiles
            WHERE id = auth.uid()
            AND user_type IN ('admin', 'super_admin')
        )
    );
```

3.2.5. Admin Management Tables

admin_tasks

```
 CREATE TABLE admin_tasks (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    assigned_to UUID REFERENCES profiles(id) ON DELETE SET NULL,
    dispute_item_id UUID REFERENCES dispute_items(id) ON DELETE CASCADE,
    task_type TEXT NOT NULL CHECK (task_type IN (
        'review_dispute', 'generate_letter', 'send_letter',
        'follow_up', 'process_response', 'client_contact'
    )),
    title TEXT NOT NULL,
    description TEXT,
    priority TEXT DEFAULT 'medium' CHECK (priority IN ('low', 'medium', 'high',
'urgent')),
    status TEXT DEFAULT 'pending' CHECK (status IN ('pending', 'in_progress',
'completed', 'cancelled')),
    due_date TIMESTAMP WITH TIME ZONE,
    completed_at TIMESTAMP WITH TIME ZONE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
```

```
);

CREATE INDEX idx_admin_tasks_assigned_to ON admin_tasks(assigned_to);
CREATE INDEX idx_admin_tasks_status ON admin_tasks(status);
CREATE INDEX idx_admin_tasks_due_date ON admin_tasks(due_date);

ALTER TABLE admin_tasks ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Admins can view assigned tasks" ON admin_tasks
  FOR SELECT USING (
    assigned_to = auth.uid() OR
    EXISTS (
      SELECT 1 FROM profiles
      WHERE id = auth.uid()
      AND user_type = 'super_admin'
    )
  );
```

letter_templates

```
 CREATE TABLE letter_templates (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name TEXT NOT NULL,
  description TEXT,
  template_content TEXT NOT NULL, -- HTML template with placeholders
  dispute_type TEXT CHECK (dispute_type IN ('account', 'inquiry', 'public_record',
'personal_info')),
  is_active BOOLEAN DEFAULT TRUE,
  version INTEGER DEFAULT 1,
  created_by UUID REFERENCES profiles(id),
  created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
  updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE INDEX idx_letter_templates_dispute_type ON
letter_templates(dispute_type);
CREATE INDEX idx_letter_templates_active ON letter_templates(is_active) WHERE
is_active = TRUE;

ALTER TABLE letter_templates ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Admins can manage letter templates" ON letter_templates
  FOR ALL USING (
    EXISTS (
      SELECT 1 FROM profiles
      WHERE id = auth.uid()
      AND user_type IN ('admin', 'super_admin')
    )
  );
```

3.2.6. Communication & Audit Tables

client_messages

```
 CREATE TABLE client_messages (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    recipient_id UUID NOT NULL REFERENCES profiles(id) ON DELETE CASCADE,
    sender_id UUID NOT NULL REFERENCES profiles(id) ON DELETE CASCADE,
    subject TEXT NOT NULL,
    message_content TEXT NOT NULL,
    message_type TEXT DEFAULT 'general' CHECK (message_type IN ('general',
'dispute_update', 'welcome', 'reminder')),
    is_read BOOLEAN DEFAULT FALSE,
    read_at TIMESTAMP WITH TIME ZONE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE INDEX idx_client_messages_recipient ON client_messages(recipient_id);
CREATE INDEX idx_client_messages_created_at ON client_messages(created_at);

ALTER TABLE client_messages ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Users can view own messages" ON client_messages
    FOR SELECT USING (recipient_id = auth.uid());
```

audit_logs

```
 CREATE TABLE audit_logs (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID REFERENCES profiles(id) ON DELETE SET NULL,
    action TEXT NOT NULL,
    table_name TEXT NOT NULL,
    record_id UUID,
    old_values JSONB,
    new_values JSONB,
    ip_address INET,
    user_agent TEXT,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE INDEX idx_audit_logs_user_id ON audit_logs(user_id);
CREATE INDEX idx_audit_logs_table_name ON audit_logs(table_name);
CREATE INDEX idx_audit_logs_created_at ON audit_logs(created_at);

-- Only super admins can view audit logs
ALTER TABLE audit_logs ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Super admins can view audit logs" ON audit_logs
    FOR SELECT USING (
```

```
        EXISTS (
            SELECT 1 FROM profiles
            WHERE id = auth.uid()
            AND user_type = 'super_admin'
        )
    );
```

3.3. Database Functions and Triggers

3.3.1. Automated Timestamp Updates

```
-- Function to update the updated_at timestamp
CREATE OR REPLACE FUNCTION update_updated_at_column()
RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at = NOW();
    RETURN NEW;
END;
$$ language 'plpgsql';

-- Apply to relevant tables
CREATE TRIGGER update_profiles_updated_at
    BEFORE UPDATE ON profiles
    FOR EACH ROW EXECUTE FUNCTION update_updated_at_column();

CREATE TRIGGER update_dispute_items_updated_at
    BEFORE UPDATE ON dispute_items
    FOR EACH ROW EXECUTE FUNCTION update_updated_at_column();

CREATE TRIGGER update_admin_tasks_updated_at
    BEFORE UPDATE ON admin_tasks
    FOR EACH ROW EXECUTE FUNCTION update_updated_at_column();
```

3.3.2. Audit Logging Triggers

```
-- Function to log changes to audit_logs table
CREATE OR REPLACE FUNCTION log_audit_changes()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO audit_logs (
        user_id, action, table_name, record_id, old_values, new_values
    ) VALUES (
        auth.uid(),
        TG_OP,
        TG_TABLE_NAME,
        COALESCE(NEW.id, OLD.id),
        CASE WHEN TG_OP = 'DELETE' THEN to_jsonb(OLD) ELSE NULL END,
        CASE WHEN TG_OP IN ('INSERT', 'UPDATE') THEN to_jsonb(NEW) ELSE NULL
```

```
END
    );

    RETURN COALESCE(NEW, OLD);
END;
$$ language 'plpgsql';

-- Apply audit logging to sensitive tables
CREATE TRIGGER audit_dispute_items
    AFTER INSERT OR UPDATE OR DELETE ON dispute_items
    FOR EACH ROW EXECUTE FUNCTION log_audit_changes();

CREATE TRIGGER audit_dispute_letters
    AFTER INSERT OR UPDATE OR DELETE ON dispute_letters
    FOR EACH ROW EXECUTE FUNCTION log_audit_changes();
```

3.3.3. Automated Task Creation

```
 -- Function to create admin tasks based on dispute status changes
CREATE OR REPLACE FUNCTION create_admin_task_on_dispute_status_change()
RETURNS TRIGGER AS $$
BEGIN
    -- Create task when dispute is approved
    IF NEW.status = 'approved' AND OLD.status = 'pending_review' THEN
        INSERT INTO admin_tasks (
            dispute_item_id, task_type, title, description, priority, due_date
        ) VALUES (
            NEW.id,
            'generate_letter',
            'Generate dispute letter for ' || (SELECT first_name || ' ' || last_name
FROM profiles WHERE id = NEW.user_id),
            'Generate and review dispute letter for approved dispute item',
            NEW.priority,
            NOW() + INTERVAL '2 days'
        );
    END IF;

    -- Create follow-up task when letter is sent
    IF NEW.status = 'letter_sent' AND OLD.status != 'letter_sent' THEN
        INSERT INTO admin_tasks (
            dispute_item_id, task_type, title, description, priority, due_date
        ) VALUES (
            NEW.id,
            'follow_up',
            'Follow up on dispute response',
            'Check for bureau response and update dispute status',
            'medium',
            NOW() + INTERVAL '30 days'
        );
    END IF;
```

```
    RETURN NEW;
END;
$$ language 'plpgsql';

CREATE TRIGGER create_admin_tasks_on_dispute_change
    AFTER UPDATE ON dispute_items
    FOR EACH ROW EXECUTE FUNCTION
create_admin_task_on_dispute_status_change();
```

4. API Design and Edge Functions

4.1. Supabase Edge Functions Architecture

Supabase Edge Functions provide the serverless compute layer for SAINTRIX Beta, handling complex business logic, third-party integrations, and secure operations that cannot be performed client-side. These functions run on Deno runtime and are deployed globally for low latency.

4.1.1. Credit Report Import Function

```
 // supabase/functions/import-credit-report/index.ts
import { serve } from "https://deno.land/std@0.168.0/http/server.ts"
import { createClient } from 'https://esm.sh/@supabase/supabase-js@2'

interface CreditReportRequest {
  userId: string
  provider: 'plaid' | 'finicity'
  accessToken: string
}

serve(async (req) => {
  try {
    const { userId, provider, accessToken }: CreditReportRequest = await req.json()

    // Initialize Supabase client with service role key
    const supabase = createClient(
      Deno.env.get('SUPABASE_URL') ?? '',
      Deno.env.get('SUPABASE_SERVICE_ROLE_KEY') ?? ''
    )

    // Verify user authentication
    const authHeader = req.headers.get('Authorization')!
    const { data: { user }, error: authError } = await supabase.auth.getUser(
      authHeader.replace('Bearer ', '')
    )
```

```
    if (authError || !user || user.id !== userId) {
      return new Response('Unauthorized', { status: 401 })
    }

    // Call third-party credit API
    const creditData = await fetchCreditReport(provider, accessToken)

    // Parse and store credit report data
    const { data: creditReport, error: insertError } = await supabase
      .from('credit_reports')
      .insert({
        user_id: userId,
        bureau_name: creditData.bureau,
        report_date: creditData.reportDate,
        credit_score: creditData.score,
        raw_data: creditData,
        import_status: 'completed'
      })
      .select()
      .single()

    if (insertError) throw insertError

    // Parse and store individual credit accounts
    await parseCreditAccounts(supabase, creditReport.id, creditData.accounts)

    // Parse and store credit inquiries
    await parseCreditInquiries(supabase, creditReport.id, creditData.inquiries)

    return new Response(
      JSON.stringify({ success: true, creditReportId: creditReport.id }),
      { headers: { "Content-Type": "application/json" } }
    )

  } catch (error) {
    console.error('Credit report import error:', error)
    return new Response(
      JSON.stringify({ error: 'Failed to import credit report' }),
      { status: 500, headers: { "Content-Type": "application/json" } }
    )
  }
})

async function fetchCreditReport(provider: string, accessToken: string) {
  // Implementation depends on chosen credit data provider
  // This is a placeholder for the actual API integration
  const apiUrl = provider === 'plaid'
    ? 'https://production.plaid.com/credit/get'
    : 'https://api.finicity.com/aggregation/v1/customers/{customerId}/accounts'

  const response = await fetch(apiUrl, {
    method: 'POST',
```

```
    headers: {
      'Authorization': `Bearer ${accessToken}`,
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      // Provider-specific request parameters
    })
  })

  if (!response.ok) {
    throw new Error(`Credit API error: ${response.statusText}`)
  }

  return await response.json()
}
```

4.1.2. Dispute Letter Generation Function

```
// supabase/functions/generate-dispute-letter/index.ts
import { serve } from "https://deno.land/std@0.168.0/http/server.ts"
import { createClient } from 'https://esm.sh/@supabase/supabase-js@2'

interface DisputeLetterRequest {
  disputeItemIds: string[]
  templateId?: string
}

serve(async (req) => {
  try {
    const { disputeItemIds, templateId }: DisputeLetterRequest = await req.json()

    const supabase = createClient(
      Deno.env.get('SUPABASE_URL') ?? '',
      Deno.env.get('SUPABASE_SERVICE_ROLE_KEY') ?? ''
    )

    // Verify admin authentication
    const authHeader = req.headers.get('Authorization')!
    const { data: { user }, error: authError } = await supabase.auth.getUser(
      authHeader.replace('Bearer ', '')
    )

    if (authError || !user) {
      return new Response('Unauthorized', { status: 401 })
    }

    // Verify admin role
    const { data: profile } = await supabase
      .from('profiles')
      .select('user_type')
```

```
    .eq('id', user.id)
    .single()

  if (!profile || !['admin', 'super_admin'].includes(profile.user_type)) {
    return new Response('Forbidden', { status: 403 })
  }

  // Fetch dispute items with related data
  const { data: disputeItems, error: fetchError } = await supabase
    .from('dispute_items')
    .select(`
      *,
      profiles!inner(*),
      credit_accounts(*),
      credit_inquiries(*),
      public_records(*)
    `)
    .in('id', disputeItemIds)

  if (fetchError) throw fetchError

  // Group dispute items by user
  const disputesByUser = disputeItems.reduce((acc, item) => {
    const userId = item.user_id
    if (!acc[userId]) acc[userId] = []
    acc[userId].push(item)
    return acc
  }, {} as Record<string, any[]>)

  const generatedLetters = []

  // Generate letter for each user
  for (const [userId, userDisputes] of Object.entries(disputesByUser)) {
    const letterContent = await generateLetterContent(
      supabase,
      userDisputes,
      templateId
    )

    // Generate PDF
    const pdfBuffer = await generatePDF(letterContent)

    // Upload PDF to Supabase Storage
    const fileName = `dispute-letter-${userId}-${Date.now()}.pdf`
    const { data: uploadData, error: uploadError } = await supabase.storage
      .from('dispute-letters')
      .upload(fileName, pdfBuffer, {
        contentType: 'application/pdf'
      })

    if (uploadError) throw uploadError
```

```
      // Save letter record to database
      for (const dispute of userDisputes) {
        const { data: letter, error: letterError } = await supabase
          .from('dispute_letters')
          .insert({
            dispute_item_id: dispute.id,
            template_id: templateId,
            letter_content: letterContent,
            letter_pdf_path: uploadData.path,
            status: 'generated'
          })
          .select()
          .single()

        if (letterError) throw letterError
        generatedLetters.push(letter)

        // Update dispute item status
        await supabase
          .from('dispute_items')
          .update({ status: 'letter_generated' })
          .eq('id', dispute.id)
      }
    }

    return new Response(
      JSON.stringify({
        success: true,
        letters: generatedLetters
      }),
      { headers: { "Content-Type": "application/json" } }
    )

  } catch (error) {
    console.error('Letter generation error:', error)
    return new Response(
      JSON.stringify({ error: 'Failed to generate dispute letter' }),
      { status: 500, headers: { "Content-Type": "application/json" } }
    )
  }
})
```

4.2. Next.js API Routes

Next.js API routes serve as the middleware layer between the frontend and Supabase, providing additional security, caching, and request processing capabilities.

4.2.1. Credit Report Import API Route

```
// pages/api/credit/import.ts
import { NextApiRequest, NextApiResponse } from 'next'
import { createServerSupabaseClient } from '@supabase/auth-helpers-nextjs'

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  if (req.method !== 'POST') {
    return res.status(405).json({ error: 'Method not allowed' })
  }

  try {
    const supabase = createServerSupabaseClient({ req, res })

    // Verify user authentication
    const { data: { session }, error: authError } = await supabase.auth.getSession()

    if (authError || !session) {
      return res.status(401).json({ error: 'Unauthorized' })
    }

    const { provider, accessToken } = req.body

    // Validate input
    if (!provider || !accessToken) {
      return res.status(400).json({ error: 'Missing required parameters' })
    }

    // Call Supabase Edge Function
    const { data, error } = await supabase.functions.invoke('import-credit-report', {
      body: {
        userId: session.user.id,
        provider,
        accessToken
      }
    })

    if (error) {
      console.error('Edge function error:', error)
      return res.status(500).json({ error: 'Failed to import credit report' })
    }

    res.status(200).json(data)

  } catch (error) {
    console.error('API route error:', error)
    res.status(500).json({ error: 'Internal server error' })
```

```
  }
}
```

5. Security Implementation

5.1. Authentication and Authorization

SAINTRIX Beta implements a comprehensive security model using Supabase Auth combined with custom authorization logic to ensure that sensitive financial data is properly protected.

Multi-Factor Authentication (MFA):

```
// Enable MFA for user accounts
const { data, error } = await supabase.auth.mfa.enroll({
  factorType: 'totp',
  friendlyName: 'SAINTRIX Beta'
})

// Verify MFA during login
const { data: verifyData, error: verifyError } = await supabase.auth.mfa.verify({
  factorId: factor.id,
  challengeId: challenge.id,
  code: userEnteredCode
})
```

Row-Level Security (RLS) Policies:

```
-- Comprehensive RLS policy for dispute items
CREATE POLICY "dispute_items_access_policy" ON dispute_items
    FOR ALL USING (
        -- Users can access their own dispute items
        user_id = auth.uid()
        OR
        -- Admins can access all dispute items
        EXISTS (
            SELECT 1 FROM profiles
            WHERE id = auth.uid()
            AND user_type IN ('admin', 'super_admin')
        )
        OR
        -- Case managers can access assigned client dispute items
        EXISTS (
            SELECT 1 FROM admin_tasks at
            JOIN profiles p ON p.id = auth.uid()
            WHERE at.dispute_item_id = dispute_items.id
```

```
        AND at.assigned_to = auth.uid()
        AND p.user_type = 'admin'
    )
  );
```

5.2. Data Encryption

Field-Level Encryption for Sensitive Data:

```
 -- Create encryption key (stored in Supabase Vault)
SELECT vault.create_secret('ssn_encryption_key', 'your-256-bit-encryption-key');

-- Function to encrypt SSN
CREATE OR REPLACE FUNCTION encrypt_ssn(ssn_plain TEXT)
RETURNS TEXT AS $$
DECLARE
  encryption_key TEXT;
BEGIN
  SELECT decrypted_secret INTO encryption_key
  FROM vault.decrypted_secrets
  WHERE name = 'ssn_encryption_key';

  RETURN encode(
    encrypt(ssn_plain::bytea, encryption_key::bytea, 'aes'),
    'base64'
  );
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;

-- Function to decrypt SSN (only for authorized operations)
CREATE OR REPLACE FUNCTION decrypt_ssn(ssn_encrypted TEXT)
RETURNS TEXT AS $$
DECLARE
  encryption_key TEXT;
BEGIN
  -- Verify user has permission to decrypt
  IF NOT EXISTS (
    SELECT 1 FROM profiles
    WHERE id = auth.uid()
    AND user_type IN ('admin', 'super_admin')
  ) THEN
    RAISE EXCEPTION 'Unauthorized decryption attempt';
  END IF;

  SELECT decrypted_secret INTO encryption_key
  FROM vault.decrypted_secrets
  WHERE name = 'ssn_encryption_key';

  RETURN convert_from(
    decrypt(decode(ssn_encrypted, 'base64'), encryption_key::bytea, 'aes'),
```

```
        'UTF8'
    );
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;
```

5.3. Input Validation and Sanitization

Comprehensive Input Validation:

```
// Zod schemas for input validation
import { z } from 'zod'

export const CreateDisputeItemSchema = z.object({
  creditAccountId: z.string().uuid().optional(),
  creditInquiryId: z.string().uuid().optional(),
  publicRecordId: z.string().uuid().optional(),
  disputeType: z.enum(['account', 'inquiry', 'public_record', 'personal_info']),
  disputeReason: z.string().min(1).max(500),
  clientNotes: z.string().max(2000).optional()
}).refine(
  (data) => {
    const refs = [data.creditAccountId, data.creditInquiryId, data.publicRecordId]
    return refs.filter(Boolean).length === 1
  },
  {
    message: "Exactly one reference ID must be provided"
  }
)

export const UpdateProfileSchema = z.object({
  firstName: z.string().min(1).max(50).regex(/^[a-zA-Z\s'-]+$/),
  lastName: z.string().min(1).max(50).regex(/^[a-zA-Z\s'-]+$/),
  phoneNumber: z.string().regex(/^\+?1?[2-9]\d{2}[2-9]\d{2}\d{4}$/).optional(),
  addressLine1: z.string().max(100).optional(),
  city: z.string().max(50).regex(/^[a-zA-Z\s'-]+$/).optional(),
  state: z.string().length(2).regex(/^[A-Z]{2}$/).optional(),
  zipCode: z.string().regex(/^\d{5}(-\d{4})?$/).optional()
})
```

6. Performance Optimization

6.1. Database Optimization

Strategic Indexing:

```sql
-- Composite indexes for common query patterns
CREATE INDEX idx_dispute_items_user_status ON dispute_items(user_id, status);
CREATE INDEX idx_credit_accounts_report_negative ON
credit_accounts(credit_report_id, is_negative);
CREATE INDEX idx_admin_tasks_assigned_status_due ON
admin_tasks(assigned_to, status, due_date);

-- Partial indexes for performance
CREATE INDEX idx_active_dispute_items ON dispute_items(created_at)
WHERE status NOT IN ('resolved', 'unresolved');

CREATE INDEX idx_pending_admin_tasks ON admin_tasks(due_date)
WHERE status = 'pending';

-- JSONB indexes for credit report data
CREATE INDEX idx_credit_reports_raw_data_score ON credit_reports
USING GIN ((raw_data->'score'));

CREATE INDEX idx_credit_accounts_payment_history ON credit_accounts
USING GIN (payment_history);
```

Query Optimization:

```sql
-- Optimized query for client dashboard
CREATE OR REPLACE VIEW client_dashboard_view AS
SELECT
    p.id as user_id,
    p.first_name,
    p.last_name,
    cr.credit_score,
    cr.report_date,
    COUNT(CASE WHEN ca.is_negative = true THEN 1 END) as negative_accounts,
    COUNT(di.id) as active_disputes,
    COUNT(CASE WHEN di.status = 'resolved' THEN 1 END) as resolved_disputes
FROM profiles p
LEFT JOIN credit_reports cr ON cr.user_id = p.id
    AND cr.report_date = (
        SELECT MAX(report_date)
        FROM credit_reports cr2
        WHERE cr2.user_id = p.id
    )
LEFT JOIN credit_accounts ca ON ca.credit_report_id = cr.id
LEFT JOIN dispute_items di ON di.user_id = p.id
    AND di.status NOT IN ('resolved', 'unresolved')
WHERE p.user_type = 'client'
GROUP BY p.id, p.first_name, p.last_name, cr.credit_score, cr.report_date;
```

6.2. Frontend Performance

Code Splitting and Lazy Loading:

```
 // Dynamic imports for route-based code splitting
import dynamic from 'next/dynamic'

const AdminDashboard = dynamic(() => import('../components/
AdminDashboard'), {
  loading: () => <DashboardSkeleton />,
  ssr: false
})

const DisputeManager = dynamic(() => import('../components/DisputeManager'), {
  loading: () => <LoadingSpinner />
})

// Lazy loading for heavy components
const CreditReportViewer = dynamic(
  () => import('../components/CreditReportViewer'),
  { ssr: false }
)
```

Optimized Data Fetching:

```
 // SWR for client-side data fetching with caching
import useSWR from 'swr'

export function useDisputeItems(userId: string) {
  const { data, error, mutate } = useSWR(
    userId ? `/api/disputes/${userId}` : null,
    fetcher,
    {
      refreshInterval: 30000, // Refresh every 30 seconds
      revalidateOnFocus: false,
      dedupingInterval: 10000
    }
  )

  return {
    disputeItems: data,
    isLoading: !error && !data,
    isError: error,
    mutate
  }
}

// React Query for server state management
import { useQuery, useMutation, useQueryClient } from '@tanstack/react-query'
```

```
export function useCreditReports(userId: string) {
  return useQuery({
    queryKey: ['creditReports', userId],
    queryFn: () => fetchCreditReports(userId),
    staleTime: 5 * 60 * 1000, // 5 minutes
    cacheTime: 10 * 60 * 1000, // 10 minutes
    enabled: !!userId
  })
}
```

Author: Manus AI

Date: July 31, 2025

Version: 1.0 Technical Architecture