VerifyLens App - Comprehensive Technical Analysis

Document Created: October 28, 2025
Repository: https://github.com/Jgabbard61/roblox-tool
Purpose: Deep analysis of current implementation to design automated credit/billing system

---

Table of Contents

Executive Summary

Current State

- **Live App:** verifylens.com (deployed on Vercel)
- **Landing Page:** www.verifylens.com (separate repository: roblox-lander)
- **Database:** PostgreSQL (hosted on Supabase)
- **Authentication:** NextAuth.js with credential-based login
- **User Management:** Manual account creation via Super Admin dashboard
- **Search Types:**
- **Exact Search** (direct username/userId lookup via `/api/roblox`)
- **Smart Search** (fuzzy matching with AI ranking via `/api/search`)
- **Credit System: NONE** - No payment, billing, or credit tracking exists
- **Manual Process:** Admin manually creates customer accounts when someone purchases

Critical Gaps

❌ No self-service account registration
❌ No payment integration (Stripe or otherwise)
❌ No credit/usage tracking system
❌ No automated credit allocation after payment
❌ No credit deduction per search

❌ No billing/subscription management
❌ Manual account creation is not scalable

Current Architecture

System Components

```
┌──────────────────────────────────────────────────────────────┐
│                    Landing Page (roblox-lander)              │
│                    www.verifylens.com                        │
│  - Marketing content                                         │
│  - Pricing page (static)                                     │
│  - Contact forms                                             │
│  - NO payment integration                                    │
└──────────────────────────────────────────────────────────────┘
                              │
                              │ User clicks "Client Login"
                              │ (manual redirect after purchase)
                              ↓
┌──────────────────────────────────────────────────────────────┐
│                    Main App (roblox-tool)                    │
│                    verifylens.com                            │
│                                                              │
│  Authentication:                                             │
│  - NextAuth.js (JWT-based)                                   │
│  - Username/Password login                                   │
│  - NO self-registration                                      │
│                                                              │
│  User Roles:                                                 │
│  - SUPER_ADMIN (platform owner)                              │
│  - CUSTOMER_ADMIN (customer organization admin)              │
│  - CUSTOMER_USER (regular user)                              │
│                                                              │
│  Search Features:                                            │
│  - Exact Match (username/userId lookup)                      │
│  - Smart Match (fuzzy search with AI ranking)                │
│  - Display Name search                                       │
│  - Forensic mode (deep profile analysis)                     │
│                                                              │
│  Admin Dashboard (SUPER_ADMIN only):                         │
│  - Customer management (CRUD)                                │
│  - User management (CRUD)                                    │
│  - Search history viewer                                     │
│  - Analytics & statistics                                    │
└──────────────────────────────────────────────────────────────┘
                              │
                              ↓
┌──────────────────────────────────────────────────────────────┐
│                    PostgreSQL Database (Supabase)            │
│                                                              │
│  Tables:                                                     │
│  - customers                                                 │
│  - users                                                     │
│  - search_history                                            │
│  - audit_logs                                                │
│                                                              │
│  Missing Tables:                                             │
│  ❌ credits / credit_transactions                            │
│  ❌ subscriptions                                            │
│  ❌ payments                                                 │
│  ❌ invoices                                                 │
└──────────────────────────────────────────────────────────────┘
```

Technology Stack

Frontend

- **Framework:** Next.js 15.5.4 (App Router)
- **Language:** TypeScript 5
- **UI Library:** React 19.1.0
- **Styling:** Tailwind CSS 3.4.17
- **Icons:** Lucide React 0.545.0
- **State Management:** React hooks + NextAuth session

Backend

- **API Routes:** Next.js API routes (App Router)
- **Authentication:** NextAuth.js 4.24.11
- **Database Client:** node-postgres (pg 8.16.3)
- **Database Pool:** Connection pooling via pg.Pool
- **Password Hashing:** bcrypt 6.0.0
- **Session Strategy:** JWT (JSON Web Tokens)

Database

- **RDBMS:** PostgreSQL
- **Hosting:** Supabase (managed PostgreSQL)
- **Schema Version:** Multiple migrations applied
- **SSL:** Enabled (production)

Infrastructure

- **Hosting:** Vercel
- **Environment:** Node.js 20
- **Rate Limiting:** Redis-based (ioredis 4.30.1)
- **CSV Parsing:** PapaParse 5.5.3 (for batch searches)

External APIs

- **Roblox API:**
- `https://users.roblox.com/v1/usernames/users` (exact username lookup)
- `https://users.roblox.com/v1/users/search` (fuzzy search)
- `https://users.roblox.com/v1/users/{userId}` (user profile)
- `https://friends.roblox.com/v1/users/{userId}/friends` (friends list)

Missing Integrations

❌ Stripe (or any payment processor)
❌ Email service (SendGrid, Resend, etc.)
❌ Analytics (Mixpanel, Amplitude, etc.)
❌ Error tracking (Sentry, etc.)

Database Schema Analysis

Current Tables

1. `customers` Table

Stores customer organizations (law firms).

```
CREATE TABLE customers (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL UNIQUE,
    is_active BOOLEAN NOT NULL DEFAULT true,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    contact_email VARCHAR(255),
    max_users INTEGER DEFAULT 5,
    logo_url TEXT -- Added via migration for custom branding
);
```

Key Points:
- Each customer is a separate organization (typically a law firm)
- `max_users` limits how many users can be created (not enforced in code)
- No credit balance or subscription info stored here
- `is_active` controls whether customer can log in

2. `users` Table

Stores individual user accounts with role-based access.

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(100) NOT NULL UNIQUE,
    password_hash VARCHAR(255) NOT NULL,
    role VARCHAR(50) NOT NULL CHECK (role IN ('SUPER_ADMIN', 'CUSTOMER_ADMIN', 'CUS-
TOMER_USER')),
    customer_id INTEGER REFERENCES customers(id) ON DELETE CASCADE,
    email VARCHAR(255),
    full_name VARCHAR(255),
    is_active BOOLEAN NOT NULL DEFAULT true,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    last_login TIMESTAMP WITH TIME ZONE,

    -- Constraint: SUPER_ADMIN has NULL customer_id
    CONSTRAINT users_customer_role_check CHECK (
        (role = 'SUPER_ADMIN' AND customer_id IS NULL) OR
        (role IN ('CUSTOMER_ADMIN', 'CUSTOMER_USER') AND customer_id IS NOT NULL)
    )
);
```

Key Points:
- Three roles with different permissions
- SUPER_ADMIN is the platform owner (customer_id is NULL)
- CUSTOMER_ADMIN can manage their organization
- CUSTOMER_USER is a regular user
- No credit balance per user

Current Authentication Flow:
1. User enters username/password
2. System looks up user in database
3. bcrypt verifies password hash
4. JWT token created with user info
5. Session stored in JWT (max 30 days or 2 hours based on “remember me”)

3. `search_history` Table

Logs every search performed by users.

```
CREATE TABLE search_history (
    id SERIAL PRIMARY KEY,
    user_id INTEGER NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    customer_id INTEGER REFERENCES customers(id) ON DELETE CASCADE, -- NULL for SU-
PER_ADMIN

    -- Search details
    search_type VARCHAR(50) NOT NULL CHECK (search_type IN ('username',
'displayName', 'userId', 'url', 'exact', 'smart')),
    search_mode VARCHAR(50) NOT NULL DEFAULT 'exact' CHECK (search_mode IN ('exact', '
smart', 'displayName')),
    search_query VARCHAR(500) NOT NULL,

    -- Roblox user details (if found)
    roblox_username VARCHAR(255),
    roblox_user_id BIGINT,
    roblox_display_name VARCHAR(255),
    has_verified_badge BOOLEAN,

    -- Result metadata
    result_data JSONB, -- Full result data
    result_count INTEGER DEFAULT 0,
    result_status VARCHAR(50) CHECK (result_status IN ('success', 'no_results', 'er-
ror')),
    error_message TEXT,

    -- Performance metrics
    response_time_ms INTEGER,

    searched_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

Key Points:
- Every search is logged (for analytics and auditing)
- Tracks both exact and smart searches
- Stores result status (success/no_results/error)
- `customer_id` can be NULL (for SUPER_ADMIN searches) - added via migration
- **NO CREDIT DEDUCTION** - searches are logged but not charged

Search Types:
- `exact` - Direct username or userId lookup
- `smart` - Fuzzy search with AI ranking
- `displayName` - Display name search

4. `audit_logs` Table

Tracks admin actions and system events (optional, minimal usage).

```sql
CREATE TABLE audit_logs (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id) ON DELETE SET NULL,
    customer_id INTEGER REFERENCES customers(id) ON DELETE SET NULL,
    action VARCHAR(100) NOT NULL,
    entity_type VARCHAR(50), -- e.g., 'customer', 'user', 'search'
    entity_id INTEGER,
    old_values JSONB,
    new_values JSONB,
    ip_address INET,
    user_agent TEXT,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

Key Points:
- Currently underutilized
- Could track admin actions like customer creation, user edits, etc.
- Not currently logging all admin actions

Database Views

Two helpful views are defined for analytics:

`customer_stats` View

```sql
CREATE OR REPLACE VIEW customer_stats AS
SELECT
    c.id,
    c.name,
    c.is_active,
    c.created_at,
    COUNT(DISTINCT u.id) as total_users,
    COUNT(DISTINCT CASE WHEN u.is_active THEN u.id END) as active_users,
    COUNT(sh.id) as total_searches,
    MAX(sh.searched_at) as last_search_at,
    MAX(u.last_login) as last_login_at
FROM customers c
LEFT JOIN users u ON c.id = u.customer_id
LEFT JOIN search_history sh ON c.id = sh.customer_id
GROUP BY c.id, c.name, c.is_active, c.created_at;
```

`user_activity` View

```sql
CREATE OR REPLACE VIEW user_activity AS
SELECT
    u.id,
    u.username,
    u.role,
    u.customer_id,
    c.name as customer_name,
    u.is_active,
    u.last_login,
    COUNT(sh.id) as total_searches,
    MAX(sh.searched_at) as last_search_at,
    COUNT(CASE WHEN sh.searched_at > CURRENT_TIMESTAMP - INTERVAL '7 days' THEN 1
END) as searches_last_7_days,
    COUNT(CASE WHEN sh.searched_at > CURRENT_TIMESTAMP - INTERVAL '30 days' THEN 1
END) as searches_last_30_days
FROM users u
LEFT JOIN customers c ON u.customer_id = c.id
LEFT JOIN search_history sh ON u.id = sh.user_id
GROUP BY u.id, u.username, u.role, u.customer_id, c.name, u.is_active, u.last_login;
```

Missing Tables (Critical)

The following tables are needed for the automated billing system:

1. `credit_packages` Table (Proposed)

Defines available credit packages for purchase.

```sql
CREATE TABLE credit_packages (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    credits INTEGER NOT NULL,
    price_cents INTEGER NOT NULL, -- Price in cents (e.g., 10000 = $100)
    is_active BOOLEAN DEFAULT true,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

-- Example: 1 credit = $100
-- Package: 10 credits = $1000, 50 credits = $5000, etc.
```

2. `customer_credits` Table (Proposed)

Tracks credit balance per customer.

```sql
CREATE TABLE customer_credits (
    id SERIAL PRIMARY KEY,
    customer_id INTEGER NOT NULL REFERENCES customers(id) ON DELETE CASCADE,
    balance INTEGER NOT NULL DEFAULT 0, -- Current credit balance
    total_purchased INTEGER NOT NULL DEFAULT 0, -- Lifetime credits purchased
    total_used INTEGER NOT NULL DEFAULT 0, -- Lifetime credits used
    last_purchase_at TIMESTAMP WITH TIME ZONE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,

    UNIQUE(customer_id)
);
```

3. `credit_transactions` Table (Proposed)

Logs all credit changes (purchases, usage, refunds).

```
CREATE TABLE credit_transactions (
    id SERIAL PRIMARY KEY,
    customer_id INTEGER NOT NULL REFERENCES customers(id) ON DELETE CASCADE,
    user_id INTEGER REFERENCES users(id) ON DELETE SET NULL,

    transaction_type VARCHAR(50) NOT NULL CHECK (transaction_type IN ('purchase', 'us-
age', 'refund', 'adjustment')),
    amount INTEGER NOT NULL, -- Positive for credit, negative for debit
    balance_before INTEGER NOT NULL,
    balance_after INTEGER NOT NULL,

    -- Reference to related entities
    search_history_id INTEGER REFERENCES search_history(id) ON DELETE SET NULL,
    payment_id VARCHAR(255), -- Stripe payment ID

    description TEXT,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

4. `stripe_payments` Table (Proposed)

Stores Stripe payment records.

```
CREATE TABLE stripe_payments (
    id SERIAL PRIMARY KEY,
    customer_id INTEGER NOT NULL REFERENCES customers(id) ON DELETE CASCADE,
    user_id INTEGER REFERENCES users(id) ON DELETE SET NULL,

    stripe_payment_intent_id VARCHAR(255) NOT NULL UNIQUE,
    stripe_customer_id VARCHAR(255),

    amount_cents INTEGER NOT NULL,
    currency VARCHAR(3) DEFAULT 'usd',
    status VARCHAR(50) NOT NULL, -- succeeded, failed, pending, refunded

    credits_purchased INTEGER NOT NULL,

    metadata JSONB, -- Additional Stripe metadata

    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

Authentication System

Current Implementation

Technology: NextAuth.js v4 with Credentials Provider

File: `src/app/lib/auth.ts`

Authentication Flow:
1. User submits username + password via `/auth/signin`

2. NextAuth calls `authorize()` function
3. Function queries database for user by username
4. Verifies password using bcrypt
5. Checks user and customer active status
6. Updates `last_login` timestamp
7. Returns user object if valid, `null` if invalid
8. NextAuth creates JWT token with user info
9. Session stored in cookie (httpOnly, secure)

Session Management:
- Strategy: JWT (not database sessions)
- Max age: 30 days (if "remember me") or 2 hours (default)
- Token contains: id, username, role, customerId, customerName
- Token is refreshed on each request

Security Features:
- ✅ Password hashing with bcrypt
- ✅ JWT tokens with secret
- ✅ HTTP-only cookies
- ✅ SSL in production
- ❌ No 2FA
- ❌ No password reset flow (admin must reset)
- ❌ No email verification

Middleware Protection:
File: `src/middleware.ts`

Protects routes based on authentication:
- Public routes: `/auth/signin`, `/api/auth/*`, `/api/health`
- Protected routes: All others require authentication
- Admin-only routes: `/admin/*` requires SUPER_ADMIN role

Middleware sets headers for API routes:
- `X-User-Id`: User ID from session
- `X-Customer-Id`: Customer ID from session (or 'null')

Current Limitations:
- ❌ No self-service registration
- ❌ No email verification
- ❌ No password reset (must contact admin)
- ❌ No account recovery flow
- ❌ No OAuth providers (Google, Microsoft, etc.)

Super Admin Dashboard

Overview

URL: `/admin`
Access: SUPER_ADMIN role only
Purpose: Manually manage customers, users, and view analytics

Features

1. Dashboard Overview Tab

Displays key metrics:
- Total customers, users, searches
- Success rate of searches
- User breakdown by role
- Top 10 customers by search volume (last 30 days)
- Recent activity (last 10 searches)

API: `GET /api/admin/stats?days=30`

2. Customer Management Tab

Features:
- View all customers with stats (users, searches)
- Create new customer with admin user
- Edit customer details (name, contact email, max users)
- Activate/Deactivate customers
- Delete customers (with cascade warning)
- Upload custom logo for customer branding

APIs:
- `GET /api/admin/customers` - List all customers
- `POST /api/admin/customers` - Create customer + admin user
- Input: customerName, adminUsername, adminPassword, adminEmail
- Creates customer record
- Creates CUSTOMER_ADMIN user linked to customer
- Returns credentials (must be saved by admin)
- `PUT /api/admin/customers` - Update customer details
- `PATCH /api/admin/customers` - Toggle active status
- `DELETE /api/admin/customers?customerId=X&force=true` - Delete customer

Current Account Creation Flow (Manual):
1. Customer purchases credits (via contact/phone/email - outside system)
2. Admin receives notification (email/Slack/etc.)
3. Admin logs into Super Admin dashboard
4. Admin clicks “Create Customer” button
5. Enters customer name, admin username, password, email
6. System creates customer + admin user in database
7. Admin manually sends credentials to customer
8. Customer logs in and can create additional users

Problems with Current Flow:
- ❌ Not scalable
- ❌ Requires manual intervention
- ❌ No way to track payment → account creation
- ❌ Credits are not tracked or allocated
- ❌ Credentials sent manually (security risk)

3. User Management Tab

Features:
- View all users with pagination (50 per page)

- Filter by customer, role, search term
- Create new users
- Edit user details (role, customer, email, name, status)
- Reset user passwords (displays new credentials)
- Delete users

APIs:
- `GET /api/admin/users?page=1&limit=50&customerId=1&role=CUSTOMER_USER&search=john`
- `POST /api/admin/users` - Create user
- `PATCH /api/admin/users` - Update user
- `DELETE /api/admin/users?userId=X`
- `POST /api/admin/users/password` - Reset password

4. Search History Tab

Features:
- View all searches with full details
- Filter by:
- Customer (including “Super Admin Searches”)
- Search type (username, userId, displayName, url, exact, smart)
- Result status (success, no_results, error)
- Date range (start and end date)
- Search term (query or username)
- Pagination (50 per page)
- Display search results, timing, and user info

API:
- `GET /api/admin/search-history?page=1&limit=50&customerId=1&searchType=username&resultStatus=success&startDate=2025-01-01&endDate=2025-12-31&search=john`

UI/UX

- Modern gradient design (purple/pink theme)
- Responsive layout (mobile, tablet, desktop)
- Loading states, error handling, success notifications
- Confirmation dialogs for destructive operations
- Modal forms for create/edit operations

Search Functionality

Search Types

1. Exact Match Search

API Route: `/api/roblox` (POST and GET)
Purpose: Direct lookup for exact username or userId
Use Case: When you know the exact username/userId

Exact Username Lookup:
- **Endpoint:** `POST /api/roblox`
- **Roblox API:** `POST https://users.roblox.com/v1/usernames/users`

- **Request Body:** `{ username: "john123", includeBanned: false }`
- **Response:** Single user object (if found) or empty
- **Search Mode:** `exact`

Exact UserId Lookup:
- **Endpoint:** `GET /api/roblox?userId=123456789`
- **Roblox API:** `GET https://users.roblox.com/v1/users/123456789`
- **Response:** User profile object
- **Search Mode:** `exact`

Characteristics:
- ✅ Fast (direct API lookup)
- ✅ No rate limiting concerns
- ✅ Free tier Roblox API
- ✅ Case-sensitive for usernames
- ❌ No fuzzy matching
- ❌ Must know exact spelling

Current Cost: FREE (no credit charged)

2. Smart Match Search

API Route: `/api/search` (GET)
Purpose: Fuzzy search with AI-powered ranking
Use Case: When username is misspelled or unknown

Smart Search Flow:
- **Endpoint:** `GET /api/search?keyword=john&limit=10&searchMode=smart`
- **Roblox API:** `GET https://users.roblox.com/v1/users/search?keyword=john&limit=10`
- **Response:** Array of matching users (up to 10)
- **AI Ranking:** Uses Levenshtein distance algorithm for similarity scoring
- **Search Mode:** `smart`

Ranking Algorithm:
File: `src/app/lib/ranking.ts`

Uses `fast-levenshtein` library to score candidates based on:
- Name similarity (Levenshtein distance)
- Account signals (verified badge, account age)
- Keyword hits
- Group overlap (if enabled)
- Profile completeness

Returns top suggestions sorted by confidence score.

Characteristics:
- ✅ Handles misspellings
- ✅ Returns multiple matches
- ✅ AI-powered ranking
- ⚠️ Rate limited by Roblox (requires caching)
- ⚠️ Slower than exact match
- ⚠️ May return false positives

Current Cost: FREE (no credit charged)

Rate Limiting & Protection:
- Circuit breaker (automatic pause if Roblox API fails)
- Request queue (prioritizes requests)
- Redis caching (reduces API calls)
- Retry logic with exponential backoff
- Cooldown timer (30 seconds between smart searches)

3. Display Name Search

API Route: `/api/search` (GET)
Purpose: Search by display name instead of username
Use Case: When only display name is known

- **Endpoint:** `GET /api/search?keyword=john&limit=10&searchMode=displayName`
- **Roblox API:** `GET https://users.roblox.com/v1/users/search?keyword=john&limit=10`
- **Search Mode:** `displayName`
- **Cooldown:** 30 seconds

Current Cost: FREE (no credit charged)

Search Logging

Every search is logged to `search_history` table:

```
logSearch({
  userId: parseInt(userId),
  customerId: customerId ? parseInt(customerId) : null,
  searchType: 'username', // or 'userId', 'displayName', 'url'
  searchMode: 'exact', // or 'smart', 'displayName'
  searchQuery: keyword,
  robloxUsername: firstResult?.name,
  robloxUserId: firstResult?.id,
  robloxDisplayName: firstResult?.displayName,
  hasVerifiedBadge: firstResult?.hasVerifiedBadge,
  resultData: { users, searchResults },
  resultCount: users.length,
  resultStatus: users.length > 0 ? 'success' : 'no_results',
  responseTimeMs: responseTime,
});
```

Logged Fields:
- User who performed search
- Customer organization
- Search type and mode
- Query string
- Result details (if found)
- Result status (success/no_results/error)
- Response time (performance tracking)

Current Usage: Analytics and auditing only - **NO CREDIT DEDUCTION**

Forensic Mode

Feature: Deep profile analysis
Purpose: Gather extensive data for legal cases

Capabilities:
- User profile snapshot
- Friends list analysis
- Group memberships
- Account age verification
- Badge collection
- Historical data

API Routes:
- `/api/profile/[userId]` - Get full user profile
- `/api/friends/[userId]` - Get friends list
- `/api/forensic/report` - Generate forensic report

Current Cost: FREE (no credit charged)

Credit/Billing System Status

Current State

Status: ❌ DOES NOT EXIST

What's Missing:
- No credit balance tracking
- No payment integration (Stripe or otherwise)
- No credit deduction per search
- No usage limits enforcement
- No subscription plans
- No invoicing system
- No billing history
- No payment methods stored

Current Process:
1. Customer contacts sales (phone/email/website form)
2. Sales quotes price based on estimated usage
3. Customer pays via invoice or credit card (outside system)
4. Admin manually creates account
5. Customer logs in and uses app
6. **NO LIMITS** - customer can perform unlimited searches
7. No tracking of actual usage vs purchased credits

Pricing Model (From Landing Page):
- **Basic Lookup:** $[TBD] per lookup
- **Smart Search:** $[TBD] per lookup
- Intended model: **1 credit = $100 per search** (both exact and smart)
- Special rule: If exact search returns no results, don't charge

---

Current Workflow Issues

Problem 1: Manual Account Creation

Issue: Admin must manually create accounts after payment

Current Flow:

```
Customer → Contact Sales → Quote → Payment (outside) →
Manual Notification → Admin Dashboard → Create Account →
Send Credentials → Customer Logs In
```

Time Consuming: 10-30 minutes per customer
Error Prone: Manual credential creation/sending
Not Scalable: Can't handle high volume

Problem 2: No Credit Tracking

Issue: No way to track how many credits customer has purchased or used

Current Reality:
- Customer pays for "X searches"
- Admin creates account
- Customer can perform UNLIMITED searches
- No enforcement of purchased amount
- No way to track usage vs payment

Business Risk:
- Revenue leakage (customer uses more than purchased)
- No upsell opportunities (can't see when credits running low)
- No historical usage data for pricing optimization

Problem 3: No Payment Integration

Issue: Payments happen outside the system

Problems:
- Manual reconciliation required
- No automatic credit allocation
- No payment history in app
- No failed payment handling
- No automatic receipts

Problem 4: No Self-Service

Issue: Customers can't sign up or purchase on their own

Current Limitations:
- Must contact sales
- Must wait for admin to create account
- Can't add more credits instantly
- Can't manage own billing

Customer Experience: Poor - requires back-and-forth communication

Problem 5: Search Cost Not Defined in Code

Issue: Business rule “1 credit = $100 per search” is not implemented

Current Code:
- Both exact and smart searches are FREE
- No credit check before search
- No credit deduction after search
- Special rule (exact search with no results = free) not implemented

---

Integration Requirements

Required Integrations

1. Stripe Payment Processing

Purpose: Accept credit card payments for credit purchases

Needed Components:
- Stripe account (verifylens.com)
- Stripe.js for frontend (PCI-compliant card handling)
- Stripe Node.js SDK for backend
- Webhook endpoint for payment events

Stripe Products to Use:
- **Payment Intents API** - One-time credit purchases
- **Checkout Sessions** - Hosted payment page (easier)
- **Customer objects** - Store customer payment info
- **Invoices** (optional) - For billing records

Environment Variables:

```
STRIPE_PUBLISHABLE_KEY=pk_live_...
STRIPE_SECRET_KEY=sk_live_...
STRIPE_WEBHOOK_SECRET=whsec_...
```

Workflow:
1. Customer clicks “Buy Credits” on pricing page
2. System creates Stripe Checkout Session
3. Customer redirected to Stripe-hosted payment page
4. Customer enters card details
5. Stripe processes payment
6. Stripe webhook notifies our backend
7. Backend creates customer account + allocates credits
8. Customer receives email with login credentials
9. Customer logs in and uses credits

2. Email Service

Purpose: Send automated emails for account creation, receipts, low credit alerts

Recommended: Resend (modern, easy to use) or SendGrid (enterprise)

Email Types Needed:
- Account creation with credentials
- Payment receipt
- Low credit warning (when balance < 5 credits)
- Zero credits notification
- Password reset (future)

Environment Variables:

```
RESEND_API_KEY=re_...
FROM_EMAIL=noreply@verifylens.com
```

3. Landing Page ↔ App Integration

Current State: Two separate Next.js apps (different repos)

Needed:
- Shared authentication (or redirect after payment)
- Payment flow on landing page
- Redirect to app after account creation
- Consistent branding

Integration Options:

Option A: Payment on Landing Page

```
Landing Page → Stripe Checkout → Webhook →
Create Account in App DB → Email Credentials →
Customer Clicks "Login" → App Login Page
```

Option B: Redirect to App for Payment

```
Landing Page → "Buy Now" → Redirect to App →
Sign Up Page → Create Account → Payment →
Allocate Credits → App Home
```

Recommendation: Option A (payment on landing page) is better UX

Proposed Architecture

High-Level Automated Flow

```
┌──────────────────────────────────────────────────────────────┐
│                    Landing Page (roblox-lander)              │
│                    www.verifylens.com                        │
│                                                              │
│  1. Customer selects credit package (e.g., 10 credits)       │
│  2. Clicks "Buy Now"                                         │
│  3. Redirected to Stripe Checkout                            │
└──────────────────────────────────────────────────────────────┘
                              │
                              ↓
┌──────────────────────────────────────────────────────────────┐
│                    Stripe Hosted Checkout                    │
│                                                              │
│  - Customer enters email, password, card details             │
│  - Stripe validates and processes payment                    │
│  - On success, redirects to success page                     │
└──────────────────────────────────────────────────────────────┘
                              │
                              ↓
┌──────────────────────────────────────────────────────────────┐
│                    Stripe Webhook                            │
│                    POST /api/stripe/webhook                  │
│                                                              │
│  Event: checkout.session.completed                           │
│  Data:                                                       │
│    - Customer email                                          │
│    - Payment amount                                          │
│    - Credits purchased                                       │
│    - Stripe payment ID                                       │
└──────────────────────────────────────────────────────────────┘
                              │
                              ↓
┌──────────────────────────────────────────────────────────────┐
│                    Automated Account Creation                │
│                    (Backend Logic)                           │
│                                                              │
│  1. Verify webhook signature (security)                      │
│  2. Check if customer already exists (by email)              │
│  3. If new:                                                  │
│     a. Create customer record                                │
│     b. Generate secure username and password                 │
│     c. Create CUSTOMER_ADMIN user                            │
│     d. Initialize customer_credits record                    │
│  4. If existing:                                             │
│     a. Add credits to existing balance                       │
│  5. Create credit_transaction record (purchase)              │
│  6. Create stripe_payments record                            │
│  7. Send welcome email with credentials                      │
└──────────────────────────────────────────────────────────────┘
                              │
                              ↓
┌──────────────────────────────────────────────────────────────┐
│                    Main App (roblox-tool)                    │
│                    verifylens.com                            │
│                                                              │
│  Customer Login:                                             │
│  - Uses credentials from email                               │
│  - Logs in successfully                                      │
│                                                              │
│  Dashboard:                                                  │
│  - Shows current credit balance                              │
│  - Shows usage history                                       │
```

```
│  - "Buy More Credits" button → landing page                    │
│                                                                 │
│  Search:                                                        │
│  1. Customer performs search (exact or smart)                   │
│  2. System checks credit balance > 0                            │
│  3. If balance = 0, show "Out of Credits" modal                 │
│  4. If balance > 0:                                             │
│     a. Execute search                                           │
│     b. Log to search_history                                    │
│     c. If exact search with no results → don't charge           │
│     d. If results found → deduct 1 credit                       │
│     e. Create credit_transaction record (usage)                 │
│     f. Update customer_credits balance                          │
│  5. Display results                                             │
│                                                                 │
│  Low Credit Alert:                                              │
│  - When balance < 5, show banner "Running Low on Credits"       │
│  - Send email notification                                      │
└─────────────────────────────────────────────────────────────────┘
```

Database Changes Needed

New Tables to Add

1. `credit_packages` - Define credit tiers (10, 50, 100 credits)
2. `customer_credits` - Track balance per customer
3. `credit_transactions` - Log all credit changes
4. `stripe_payments` - Store payment records

New Columns to Add

- `customers.stripe_customer_id` - Link to Stripe customer
- `users.email_verified` - Track email verification status
- `users.password_reset_token` - For self-service password reset

New Indexes Needed

```sql
CREATE INDEX idx_customer_credits_customer_id ON customer_credits(customer_id);
CREATE INDEX idx_credit_transactions_customer_id ON credit_transactions(customer_id);
CREATE INDEX idx_credit_transactions_created_at ON credit_transactions(created_at
DESC);
CREATE INDEX idx_stripe_payments_customer_id ON stripe_payments(customer_id);
CREATE INDEX idx_stripe_payments_intent_id ON stripe_payments(stripe_payment_intent_id
);
```

API Routes to Add

Stripe Integration

- `POST /api/stripe/create-checkout-session` - Create Stripe checkout session
- `POST /api/stripe/webhook` - Handle Stripe webhooks (payment success)
- `GET /api/stripe/payment-status?sessionId=X` - Check payment status

Credit Management

- `GET /api/credits/balance` - Get current credit balance
- `GET /api/credits/transactions?page=1&limit=50` - Get credit transaction history
- `POST /api/credits/purchase` - Redirect to Stripe checkout

Self-Service Account

- `POST /api/auth/register` - Self-service registration (create account)
- `POST /api/auth/verify-email` - Email verification
- `POST /api/auth/forgot-password` - Request password reset
- `POST /api/auth/reset-password` - Reset password with token

Code Changes Needed

1. Search API Routes

File: `src/app/api/search/route.tsx` and `src/app/api/roblox/route.tsx`

Changes:

```
// Before executing search
const creditBalance = await getCustomerCreditBalance(customerId);

if (creditBalance <= 0) {
  return NextResponse.json({
    error: 'insufficient_credits',
    message: 'You have run out of credits. Please purchase more.'
  }, { status: 402 }); // 402 Payment Required
}

// Execute search...

// After getting results
if (resultCount > 0 || searchMode === 'smart') {
  // Deduct 1 credit
  await deductCredit({
    customerId,
    userId,
    searchHistoryId: loggedSearch.id,
    description: `${searchMode} search for "${searchQuery}"`
  });
}

// Special rule: Exact search with no results = free
if (searchMode === 'exact' && resultCount === 0) {
  // Don't deduct credit
}
```

2. Frontend Components

File: `src/app/page.tsx`

Changes:
- Add credit balance display in header
- Show “Insufficient Credits” modal when balance = 0
- Add “Buy More Credits” button
- Add low credit warning banner

New Components:
- `components/CreditBalanceDisplay.tsx`
- `components/InsufficientCreditsModal.tsx`
- `components/LowCreditBanner.tsx`
- `components/BuyCreditsButton.tsx`

3. Database Utilities

File: `src/app/lib/db/index.ts`

New Functions:

```
// Get customer credit balance
export async function getCustomerCreditBalance(customerId: number): Promise<number>

// Deduct credit from customer
export async function deductCredit(params: {
  customerId: number;
  userId: number;
  searchHistoryId: number;
  description?: string;
}): Promise<void>

// Add credits to customer (after payment)
export async function addCredits(params: {
  customerId: number;
  amount: number;
  paymentId: string;
  description?: string;
}): Promise<void>

// Get credit transaction history
export async function getCreditTransactions(
  customerId: number,
  page: number,
  limit: number
): Promise<CreditTransaction[]>
```

Environment Variables to Add

App (roblox-tool):

```
# Stripe
STRIPE_PUBLISHABLE_KEY=pk_live_...
STRIPE_SECRET_KEY=sk_live_...
STRIPE_WEBHOOK_SECRET=whsec_...

# Email
RESEND_API_KEY=re_...
FROM_EMAIL=noreply@verifylens.com

# App URLs
NEXT_PUBLIC_APP_URL=https://verifylens.com
NEXT_PUBLIC_LANDING_PAGE_URL=https://www.verifylens.com
```

Landing Page (roblox-lander):

```
# Stripe
NEXT_PUBLIC_STRIPE_PUBLISHABLE_KEY=pk_live_...
STRIPE_SECRET_KEY=sk_live_...

# App URL
NEXT_PUBLIC_APP_URL=https://verifylens.com
```

Implementation Phases

Phase 1: Database & Backend Foundation

Duration: 1-2 days

Tasks:
- [ ] Create new database tables (credit_packages, customer_credits, credit_transactions, stripe_payments)
- [ ] Add database migration scripts
- [ ] Implement credit management functions in `lib/db`
- [ ] Add unit tests for credit logic

Deliverables:
- Migration SQL files
- Database schema updated
- Credit management functions working

Phase 2: Stripe Integration

Duration: 2-3 days

Tasks:
- [ ] Set up Stripe account
- [ ] Create Stripe products for credit packages
- [ ] Implement Stripe Checkout Session creation
- [ ] Implement Stripe webhook handler
- [ ] Test payment flow in Stripe test mode
- [ ] Add payment records to database

Deliverables:
- Stripe checkout working
- Webhook handling payment success
- Credits automatically allocated after payment

Phase 3: Automated Account Creation

Duration: 2-3 days

Tasks:
- [ ] Implement account creation logic in webhook
- [ ] Generate secure random credentials
- [ ] Email integration (Resend or SendGrid)
- [ ] Create welcome email template
- [ ] Test full flow: payment → account → email

Deliverables:
- Accounts created automatically on payment
- Credentials emailed to customer
- Customer can log in immediately

Phase 4: Credit Deduction Logic

Duration: 2-3 days

Tasks:
- [ ] Update search API routes to check credit balance
- [ ] Implement credit deduction after successful searches
- [ ] Implement special rule: exact search with no results = free
- [ ] Add error handling for insufficient credits
- [ ] Add credit transaction logging

Deliverables:
- Searches deduct credits correctly
- Special rule enforced
- Transaction history accurate

Phase 5: Frontend UI Updates

Duration: 3-4 days

Tasks:
- [ ] Add credit balance display in header
- [ ] Create “Insufficient Credits” modal
- [ ] Create low credit warning banner
- [ ] Add “Buy More Credits” button → landing page
- [ ] Create credit transaction history page
- [ ] Update user dashboard with credit info

Deliverables:
- Clean UI showing credit balance
- Users can see their usage history
- Easy path to purchase more credits

Phase 6: Landing Page Updates

Duration: 2-3 days

Tasks:
- [ ] Add “Buy Now” buttons to pricing tiers
- [ ] Implement Stripe Checkout Session creation on landing page
- [ ] Add payment success page
- [ ] Add payment failure handling
- [ ] Update pricing page with credit details

Deliverables:
- Customers can purchase credits from landing page
- Smooth payment experience
- Clear messaging about credit system

Phase 7: Testing & QA

Duration: 3-4 days

Tasks:
- [ ] End-to-end testing: purchase → account → login → search → credit deduction
- [ ] Test edge cases: payment failure, webhook retry, duplicate payments
- [ ] Test special rules: exact search with no results
- [ ] Test low credit warnings
- [ ] Test credit balance display

- [ ] Security testing: webhook signature validation, credit tampering prevention
- [ ] Performance testing: database queries under load

Deliverables:
- All flows tested and working
- Edge cases handled
- Security verified

Phase 8: Deployment & Monitoring

Duration: 1-2 days

Tasks:
- [ ] Deploy database migrations to production
- [ ] Deploy app updates to Vercel
- [ ] Deploy landing page updates to Vercel
- [ ] Configure Stripe webhook in production
- [ ] Set up monitoring and alerts
- [ ] Create admin dashboard for monitoring credit usage
- [ ] Document new processes

Deliverables:
- Fully automated system live
- Monitoring in place
- Documentation complete

Total Estimated Time: 16-24 days (3-4 weeks)

Recommended Next Steps

1. **Review this analysis** with stakeholders
2. **Finalize pricing** (1 credit = $100, or different?)
3. **Define credit packages** (10, 50, 100 credits?)
4. **Set up Stripe account**
5. **Decide on email service** (Resend recommended)
6. **Create detailed implementation plan**
7. **Start with Phase 1** (database foundation)

Appendix: Key Files Reference

Authentication

- `src/app/lib/auth.ts` - NextAuth configuration
- `src/app/api/auth/[...nextauth]/route.ts` - Auth handler
- `src/middleware.ts` - Route protection

Database

- `database/schema.sql` - Complete schema
- `src/app/lib/db/index.ts` - Database utilities

Search APIs

- `src/app/api/search/route.tsx` - Smart/fuzzy search
- `src/app/api/roblox/route.tsx` - Exact search

Admin Dashboard

- `src/app/admin/page.tsx` - Admin dashboard main page
- `src/app/admin/components/CustomerManagement.tsx` - Customer CRUD
- `src/app/admin/components/UserManagement.tsx` - User CRUD
- `src/app/api/admin/customers/route.ts` - Customer API
- `src/app/api/admin/users/route.ts` - User API
- `src/app/api/admin/stats/route.ts` - Dashboard stats

Frontend

- `src/app/page.tsx` - Main search page
- `src/app/components/SearchModeSelector.tsx` - Search mode toggle

Document End

This analysis provides a complete understanding of the current VerifyLens app architecture and a clear roadmap for implementing the automated credit/billing system. All gaps have been identified and solutions proposed.