# CouponSwap

## Technical Design Document

| | |
|---|---|
| **Project:** | CouponSwap - P2P Coupon Exchange Platform |
| **Version:** | 1.0 |
| **Date:** | January 31, 2026 |
| **Architecture:** | Serverless JAMStack (Next.js + Supabase) |
| **Target Platform:** | Web (Mobile-Responsive) |

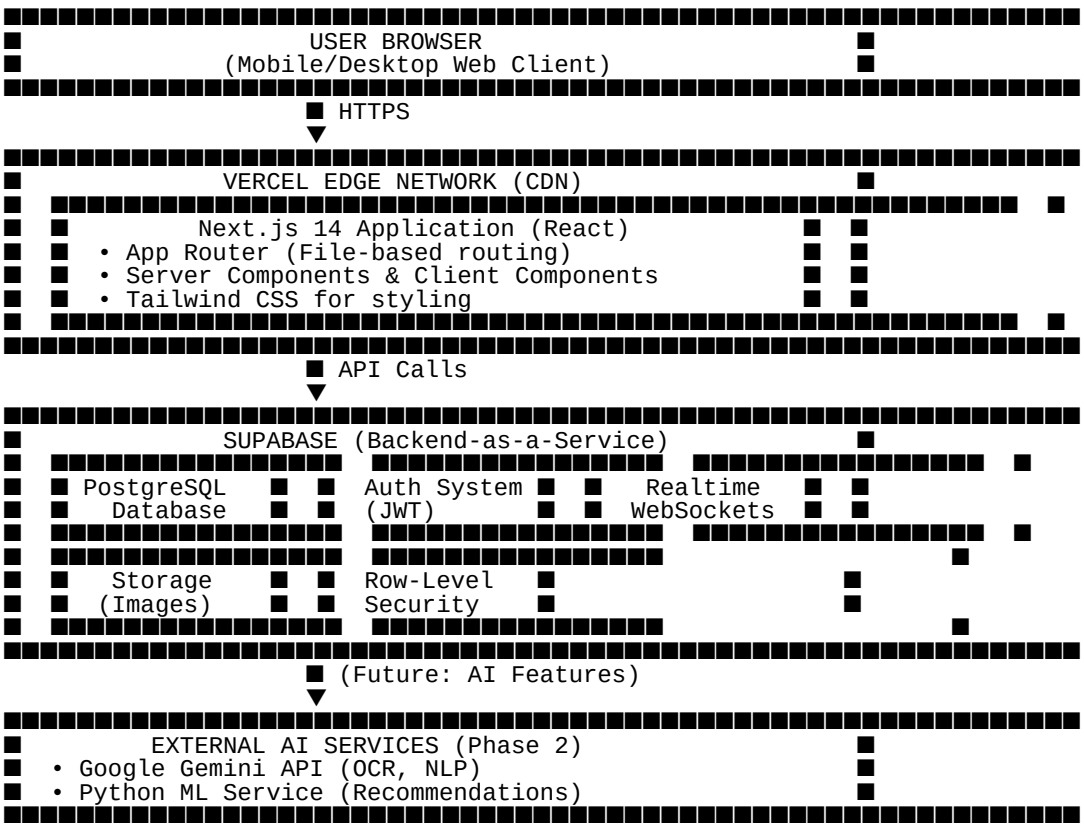
# 1. System Architecture Overview

## 1.1 High-Level Architecture

CouponSwap follows a modern serverless architecture pattern using the JAMStack approach. The frontend is a Next.js 14 application with App Router, deployed on Vercel's edge network. The backend leverages Supabase as a Backend-as-a-Service (BaaS), providing PostgreSQL database, authentication, real-time subscriptions, and storage capabilities. This architecture eliminates the need for traditional server management and scales automatically based on demand.

| Layer | Technology | Purpose |
|---|---|---|
| Presentation | Next.js 14 (React), Tailwind CSS | UI/UX, Client-side routing, SSR/SSG |
| API/Logic | Next.js Server Actions, Edge Functions | Business logic, Transaction handling |
| Backend Services | Supabase | Database, Auth, Real-time, Storage |
| Database | PostgreSQL (Supabase) | Persistent data storage |
| Hosting | Vercel (Frontend), Supabase (Backend) | CDN, Edge deployment |
| AI/ML | Google Gemini API, Python scripts | OCR, Recommendations, Fraud detection |

## 1.2 Architecture Diagram (Text Representation)



# 2. Database Schema Design

## 2.1 Entity-Relationship Overview

The database follows a relational model with four core entities: Profiles (users), Coupons (marketplace items), Transactions (credit transfers), and Messages (chat). All tables use UUIDs for primary keys and include timestamps for audit trails. Row-Level Security (RLS) policies enforce authorization at the database level.

## 2.2 Table Schemas

### profiles - User account information

| Column | Type | Constraints | Description |
|---|---|---|---|
| id | UUID | PRIMARY KEY, FK to auth.users | Unique user identifier |
| email | TEXT | | User email address |
| phone | TEXT | NULLABLE | Phone number (for OTP KYC) |
| kyc_id_number | TEXT | NULLABLE | Government ID (Aadhar/Passport) |
| kyc_verified | BOOLEAN | DEFAULT false | KYC verification status |
| credits | INTEGER | DEFAULT 50, NOT NULL | Virtual currency balance |
| created_at | TIMESTAMPTZ | DEFAULT now() | Account creation timestamp |

### coupons - Marketplace listings

| Column | Type | Constraints | Description |
|---|---|---|---|
| id | UUID | PRIMARY KEY | Unique coupon identifier |
| seller_id | UUID | FK to profiles(id), NOT NULL | Listing owner |
| title | TEXT | NOT NULL | Coupon headline (e.g., "50% off Pizza") |
| description | TEXT | NULLABLE | Additional details |
| code | TEXT | NOT NULL | Secret coupon code (hidden until bought) |
| price_credits | INTEGER | NOT NULL | Cost in virtual credits |
| category | TEXT | NULLABLE | Food/Fashion/Travel/Tech/Other |
| expiry_date | DATE | NULLABLE | Coupon expiration date |
| image_url | TEXT | NULLABLE | Screenshot/image URL |
| is_sold | BOOLEAN | DEFAULT false | Purchase status |
| buyer_id | UUID | FK to profiles(id), NULLABLE | Purchaser (null if unsold) |
| created_at | TIMESTAMPTZ | DEFAULT now() | Listing creation time |

### transactions - Credit transfer history

| Column | Type | Constraints | Description |
|---|---|---|---|
| id | UUID | PRIMARY KEY | Unique transaction ID |
| buyer_id | UUID | FK to profiles(id) | Credit spender |
| seller_id | UUID | FK to profiles(id) | Credit recipient |
| amount_credits | INTEGER | NOT NULL | Credits transferred |
| coupon_id | UUID | FK to coupons(id) | Associated coupon |
| created_at | TIMESTAMPTZ | DEFAULT now() | Transaction timestamp |

### messages - Chat conversations

| Column | Type | Constraints | Description |
|---|---|---|---|
| id | UUID | PRIMARY KEY | Unique message ID |
| sender_id | UUID | FK to profiles(id), NOT NULL | Message sender |
| receiver_id | UUID | FK to profiles(id), NOT NULL | Message recipient |
| content | TEXT | NOT NULL | Message text |
| created_at | TIMESTAMPTZ | DEFAULT now() | Send timestamp |

# 3. Security & Authentication

## 3.1 Authentication Flow

Supabase Auth handles all authentication using JWT (JSON Web Tokens). Users can sign up with email/password, and the system automatically creates a corresponding profile in the public.profiles table via a database trigger. Sessions are managed client-side with secure HTTP-only cookies.

```
1. User submits signup form (email, password, KYC ID)
2. Supabase Auth creates user in auth.users table
3. Database trigger fires → Creates profile in public.profiles
4. User receives email verification link
5. Upon verification → JWT token generated
6. Token stored in browser (HTTP-only cookie)
7. Every API request includes JWT in Authorization header
8. Supabase validates token and returns user data
```

## 3.2 Row-Level Security (RLS) Policies

RLS ensures users can only access data they own or are authorized to view. Policies are enforced at the database level, making it impossible to bypass even with direct SQL access.

| Table | Policy Name | Rule |
|---|---|---|
| coupons | view_unsold | SELECT allowed if is_sold = false |
| coupons | insert_own | INSERT allowed if auth.uid() = seller_id |
| coupons | update_own | UPDATE allowed if auth.uid() = seller_id |
| messages | view_own | SELECT allowed if auth.uid() = sender_id OR receiver_id |
| messages | insert_own | INSERT allowed if auth.uid() = sender_id |
| transactions | view_own | SELECT allowed if auth.uid() = buyer_id OR seller_id |

# 4. API Design & Endpoints

## 4.1 RESTful Conventions

The application uses Supabase's auto-generated REST API for CRUD operations and Next.js Server Actions for complex transactions. All endpoints require JWT authentication except public marketplace listings.

| Endpoint | Method | Description | Auth Required |
|---|---|---|---|
| /api/auth/signup | POST | Create new user account | No |
| /api/auth/login | POST | Authenticate user | No |
| /api/auth/logout | POST | Invalidate session | Yes |
| /api/coupons | GET | List all unsold coupons | No |
| /api/coupons | POST | Create new coupon listing | Yes |
| /api/coupons/:id | GET | Get coupon details | No |
| /api/coupons/:id/buy | POST | Purchase coupon (transaction) | Yes |
| /api/profile | GET | Get current user profile | Yes |
| /api/profile/credits | GET | Get credit balance | Yes |
| /api/messages | GET | Get chat messages | Yes |
| /api/messages | POST | Send new message | Yes |

## 4.2 Critical Transaction Logic: Buy Coupon

The purchase flow is the most critical operation, requiring ACID compliance to prevent credit loss or double-spending. This is implemented as a database transaction using Supabase's PostgreSQL transaction support.

```
FUNCTION buyCoupon(couponId, buyerId):
  BEGIN TRANSACTION

  1. LOCK coupon row (SELECT FOR UPDATE)
  2. Verify coupon is NOT sold (is_sold = false)
  3. Get buyer's credit balance
  4. Verify buyer has enough credits
  5. Get seller_id and price from coupon
  6. Deduct credits from buyer (-price)
  7. Add credits to seller (+price)
  8. Update coupon: is_sold = true, buyer_id = buyerId
  9. Insert transaction record
  10. COMMIT TRANSACTION
  11. Return coupon code to buyer

  IF any step fails → ROLLBACK TRANSACTION
```

# 5. Frontend Architecture (Next.js)

## 5.1 Folder Structure

```
coupon-swap/
├── app/                        # Next.js 14 App Router
│   ├── layout.tsx              # Root layout (Navbar, global styles)
│   ├── page.tsx                # Home/Marketplace page
│   ├── login/
│   │   └── page.tsx            # Login/Signup page
│   ├── sell/
│   │   └── page.tsx            # Create coupon listing page
│   ├── profile/
│   │   └── page.tsx            # User dashboard
│   └── chat/
│       └── [userId]/
│           └── page.tsx        # Chat interface (dynamic route)
├── components/                 # Reusable UI components
│   ├── Navbar.tsx              # Navigation bar
│   ├── CouponCard.tsx          # Marketplace item card
│   ├── BuyButton.tsx           # Purchase action component
│   └── ChatMessage.tsx         # Chat bubble component
├── lib/
│   ├── supabase.ts             # Supabase client initialization
│   └── utils.ts                # Helper functions
├── actions/
│   └── buyCoupon.ts            # Server Action for transactions
├── public/
│   └── images/                 # Static assets
├── .env.local                  # Environment variables (not committed)
├── package.json                # Dependencies
└── next.config.js              # Next.js configuration
```

## 5.2 State Management

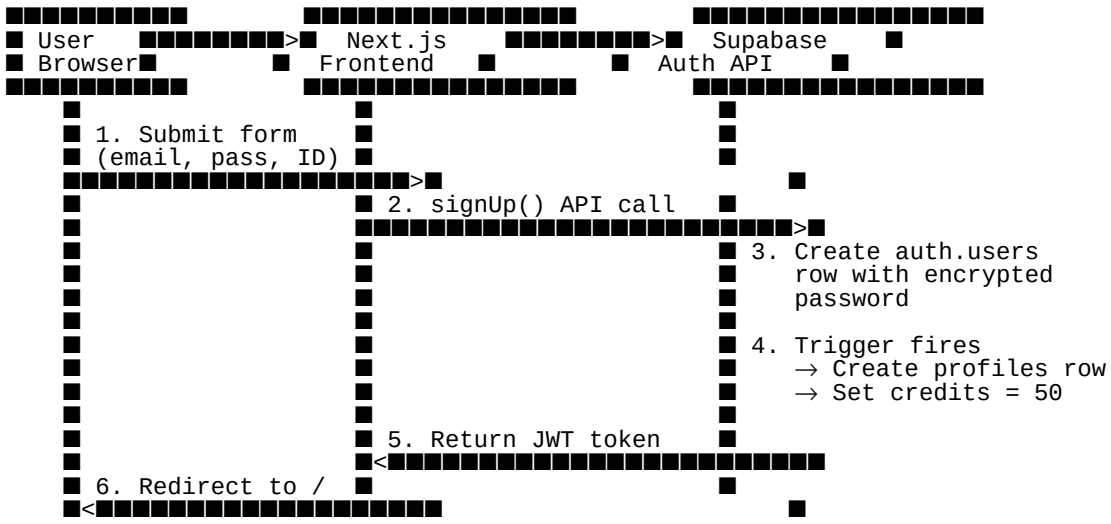
The application uses React's built-in hooks (useState, useEffect) for local component state and Supabase's real-time subscriptions for live data updates (chat messages, credit balance). No external state management library (Redux, Zustand) is needed for the MVP, keeping complexity low.

## 5.3 Real-Time Features (Chat)

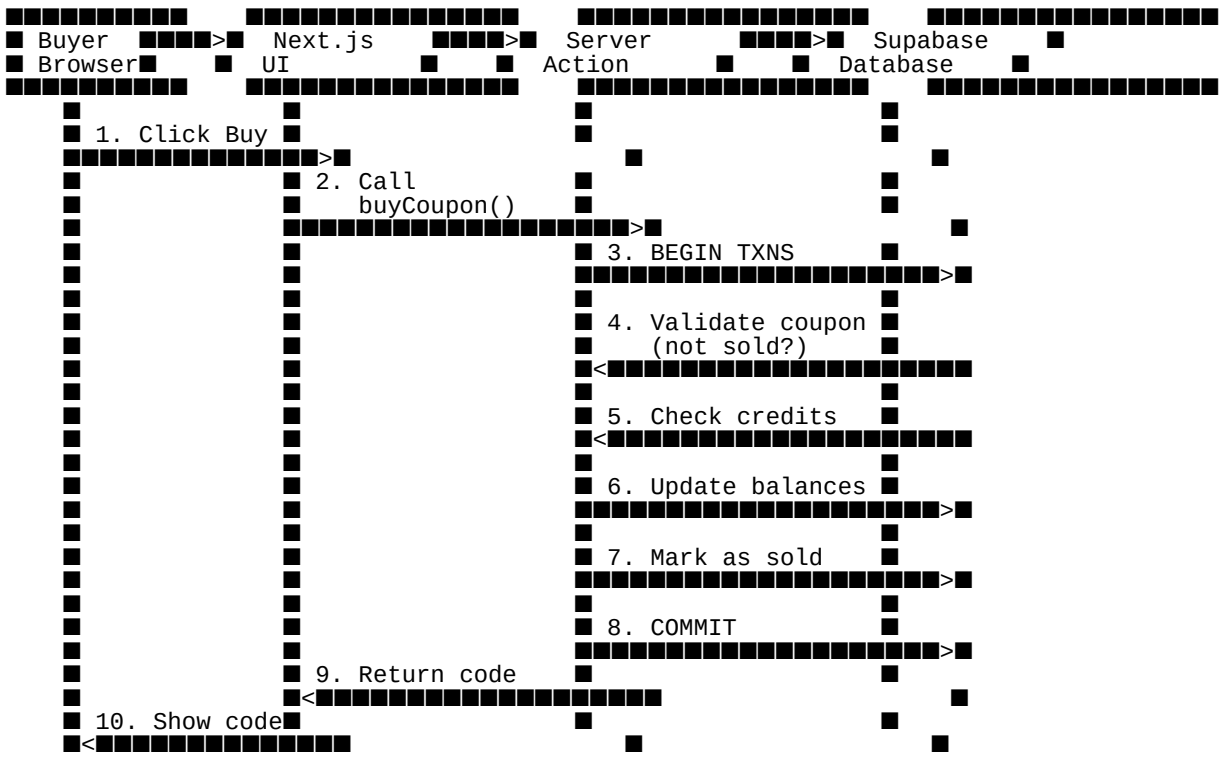
Supabase Realtime uses WebSocket connections to push database changes instantly to connected clients. The chat feature subscribes to the 'messages' table, filtering for messages where the current user is either sender or receiver.

# 6. Data Flow Diagrams

## 6.1 User Signup Flow




## 6.2 Coupon Purchase Flow



## 7. Performance Optimization Strategies

### Database Indexing

• Index on coupons.is_sold for fast unsold filtering

• Index on coupons.category for category searches

• Index on coupons.created_at for sorting

• Composite index on (is_sold, expiry_date) for expiring coupons query

### Caching (Vercel Edge)

• Static generation of landing page

• Incremental Static Regeneration (ISR) for marketplace (revalidate every 60s)

• Edge caching for public API endpoints

• Client-side caching with SWR for frequently accessed data

### Image Optimization

• Use Next.js Image component for automatic optimization

• Convert uploads to WebP format

• Lazy loading for below-the-fold images

• CDN delivery via Supabase Storage

### Code Splitting

• Automatic route-based code splitting (Next.js default)

• Dynamic imports for heavy components (chat interface)

• Tree-shaking of unused Tailwind classes

# 8. Error Handling & Validation

## 8.1 Frontend Validation

- **Email:** Must match standard email regex pattern
- **Password:** Minimum 8 characters, 1 uppercase, 1 number
- **Coupon Price:** Must be positive integer (1-1000 credits)
- **Expiry Date:** Must be future date
- **Code Field:** Required, max 50 characters

## 8.2 Error Response Format

All API errors return a consistent JSON structure with HTTP status codes. The frontend displays user-friendly messages while logging technical details to the console for debugging.

```
{
  "success": false,
  "error": {
    "code": "INSUFFICIENT_CREDITS",
    "message": "You need 50 credits to buy this coupon. Your balance: 30",
    "statusCode": 400
  }
}
```

## 8.3 Common Error Codes

| Code | HTTP Status | Meaning | Action |
|---|---|---|---|
| AUTH_REQUIRED | 401 | No valid JWT token | Redirect to login |
| INSUFFICIENT_CREDITS | 400 | Not enough credits | Prompt to buy credits |
| COUPON_SOLD | 409 | Already purchased | Refresh listing |
| INVALID_COUPON | 404 | Coupon not found | Return to marketplace |
| RATE_LIMIT | 429 | Too many requests | Show retry timer |

## 9. Testing Strategy

### Unit Testing (Jest + React Testing Library)

• Test individual components (CouponCard, BuyButton)

• Test utility functions (credit calculation, date formatting)

• Mock Supabase client for isolated testing

• Target: 70% code coverage

### Integration Testing

• Test complete user flows (signup → list → buy)

• Test database triggers and RLS policies

• Test real-time chat message delivery

• Use Supabase test project for safe testing

### Manual QA Checklist

• Cross-browser testing (Chrome, Firefox, Safari)

• Mobile responsiveness (iOS Safari, Android Chrome)

• Accessibility audit (keyboard navigation, screen readers)

• Performance testing (Lighthouse score > 90)

# 10. Deployment & DevOps

## 10.1 CI/CD Pipeline (GitHub + Vercel)

```
1. Developer pushes code to GitHub
   ■■> GitHub Actions (optional): Run linting and tests

2. Vercel detects new commit
   ■■> Automatically builds Next.js app
   ■■> Runs 'npm run build'
   ■■> Injects environment variables from Vercel dashboard

3. Preview deployment created for PR branches
   ■■> Unique URL for testing (e.g., app-pr-42.vercel.app)

4. After merge to main branch
   ■■> Production deployment triggered
   ■■> Zero-downtime deployment to couponswap.vercel.app
   ■■> Rollback available if errors detected
```

## 10.2 Monitoring & Logging

- **Vercel Analytics:** Track page views, performance metrics, Web Vitals
- **Supabase Logs:** Monitor database queries, slow queries, error rates
- **Browser Console:** Client-side error tracking (consider Sentry for production)
- **Uptime Monitoring:** Use UptimeRobot or similar for availability alerts

## 10.3 Backup & Disaster Recovery

- **Database Backups:** Supabase provides automatic daily backups (7-day retention)
- **Point-in-Time Recovery:** Available on Supabase Pro plan (upgrade if critical)
- **Code Repository:** Git history serves as code backup
- **Recovery Time Objective (RTO):** < 1 hour (redeploy from Git)
- **Recovery Point Objective (RPO):** < 24 hours (last database backup)

## 11. Scalability Considerations

While the MVP is designed for 10,000 users, the architecture supports horizontal scaling. Here's how the system can grow:

| Users | Database | Frontend | Cost/Month |
|---|---|---|---|
| 0-10K | Supabase Free Tier | Vercel Hobby (Free) | $0 |
| 10K-100K | Supabase Pro ($25) | Vercel Pro ($20) | $45 |
| 100K-1M | Dedicated PostgreSQL instance | Vercel Enterprise | $500+ |
| 1M+ | Multi-region DB, Read replicas | Edge caching, CDN | $2000+ |

### Bottleneck Mitigation

- **Database Queries:** Add connection pooling (PgBouncer), implement read replicas
- **Real-time Connections:** Horizontal scaling of Supabase Realtime servers
- **Image Storage:** Migrate to dedicated CDN (Cloudinary, ImageKit) if needed
- **API Rate Limits:** Implement Redis caching layer for frequent queries

# 12. Technical Appendix

## 12.1 Technology Versions

| Technology | Version | Purpose |
|---|---|---|
| Next.js | 14.x (App Router) | React framework |
| React | 18.x | UI library |
| TypeScript | 5.x | Type safety |
| Tailwind CSS | 3.x | Styling |
| Supabase JS Client | 2.x | Backend SDK |
| PostgreSQL | 15.x | Database |
| Node.js | 20 LTS | Runtime |

## 12.2 Key Design Decisions

### Why Next.js over Create React App?

Built-in SSR/SSG, better SEO, superior performance, easier deployment

### Why Supabase over Firebase?

PostgreSQL (relational DB), better for complex queries, open-source, SQL-based

### Why Credits instead of Real Money?

Avoids payment gateway complexity, reduces legal liability, gamification

### Why not mobile-native app?

Web-first allows faster iteration, works on all devices, no app store approval delays