Local Ticket Marketplace - Optimized Implementation Plan

Project Overview

Objective: Build a peer-to-peer marketplace for digital event tickets with automated delivery, fraud prevention, and zero initial costs.

Core Features:

1. Digital ticket upload and secure storage
2. Structured offer negotiation (no free text)
3. Automated payment processing and ticket delivery
4. Seller payouts via Stripe Connect
5. Fraud prevention and verification

Optimized Tech Stack:

- **Monorepo**: Single repository with shared types
- **Database**: Supabase CLI (PostgreSQL + Auth + Storage) - works offline!
- **ORM**: Prisma for type-safe database access
- **Backend**: Next.js API Routes (simpler than separate Express)
- **Frontend**: Next.js 14 + TypeScript + Tailwind + shadcn/ui
- **Validation**: Zod schemas (shared between frontend/backend)
- **State Management**: TanStack Query for caching
- **Image Processing**: Sharp for watermarking
- **Payments**: Stripe Connect (marketplace model)
- **Hosting**: Vercel (frontend + API) - generous free tier

PHASE 0: Minimal Working Prototype

Objective: Build the simplest possible working ticket marketplace to validate the concept before adding infrastructure complexity.

Instructions for Claude Code:

Create a single Next.js application with everything in one place:

PROJECT SETUP (10 minutes):

```
npx create-next-app@latest ticket-marketplace --typescript --tailwind --app
cd ticket-marketplace
npm install @prisma/client prisma zod react-hook-form @hookform/resolvers
npm install @supabase/supabase-js @supabase/auth-helpers-nextjs
npm install lucide-react react-hot-toast
```

SIMPLE DATABASE (SQLite for true zero-config):

```
npx prisma init --datasource-provider sqlite
```

Create this minimal schema in prisma/schema.prisma:

```
model User {
  id        String   @id @default(cuid())
  email     String   @unique
  username  String   @unique
  password  String   // Hash with bcrypt
  createdAt DateTime @default(now())

  listings  Listing[]
  offers    Offer[]
}

model Listing {
  id          String   @id @default(cuid())
  userId      String
  user        User     @relation(fields: [userId], references: [id])
  title       String
  eventName   String
  eventDate   DateTime
  price       Int      // Store in cents
  quantity    Int
  ticketPath  String?  // Local file path
  status      String   @default("active")
  createdAt   DateTime @default(now())

  offers      Offer[]
}

model Offer {
  id          String   @id @default(cuid())
  listingId   String
  listing     Listing  @relation(fields: [listingId], references: [id])
  buyerId     String
  buyer       User     @relation(fields: [buyerId], references: [id])
```

```
  offerPrice  Int      // In cents
  quantity    Int
  message     String   // Template ID
  status      String   @default("pending")
  createdAt   DateTime @default(now())
}

Run: npx prisma db push

SIMPLE FILE STRUCTURE:
app/
├── api/
│   ├── auth/[...route]/route.ts    # Login/register
│   ├── listings/route.ts            # CRUD listings
│   ├── offers/route.ts              # Make/respond to offers
│   └── upload/route.ts              # File upload
├── auth/
│   ├── login/page.tsx
│   └── register/page.tsx
├── listings/
│   ├── page.tsx                     # Browse all
│   ├── [id]/page.tsx               # View one
│   └── create/page.tsx             # Create new
├── dashboard/
│   └── page.tsx                    # User's listings/offers
├── lib/
│   ├── auth.ts                     # Simple JWT auth
│   ├── db.ts                       # Prisma client
│   └── validations.ts              # Zod schemas
├── components/
│   ├── Navbar.tsx
│   └── OfferModal.tsx
├── layout.tsx
└── page.tsx                        # Homepage

CORE FEATURES TO BUILD:

1. Simple auth (lib/auth.ts):
   - Hash passwords with bcrypt
   - Store user ID in JWT cookie
   - Basic middleware to check auth

2. File upload (api/upload/route.ts):
   - Accept PDF/images only
   - Save to public/uploads/[userId]/[filename]
   - Return file path
```

```
3. Listing CRUD:
   - Create listing with ticket upload
   - View all listings (no pagination yet)
   - View single listing

4. Offer system (keep it simple):
   - 3 message templates only:
     * "I'll buy at asking price"
     * "I offer $X"
     * "Is this still available?"
   - Seller can accept/reject
   - No complex state machine

5. Mock payment flow:
   - When offer accepted, show "Pay Now" button
   - Fake payment form
   - Mark as "completed" after mock payment
   - Show download link to buyer

WHAT TO SKIP FOR NOW:
- No Supabase (use SQLite)
- No RLS policies
- No service abstractions
- No monorepo
- No real payments
- No email notifications
- No image processing/watermarks
- No complex caching
- No rate limiting

SIMPLE VALIDATIONS:
// lib/validations.ts
import { z } from 'zod';

export const createListingSchema = z.object({
  title: z.string().min(3).max(100),
  eventName: z.string().min(3),
  eventDate: z.string().datetime(),
  price: z.number().min(100), // Minimum $1.00
  quantity: z.number().min(1).max(10),
});

export const createOfferSchema = z.object({
  listingId: z.string(),
  offerPrice: z.number().min(100),
  quantity: z.number().min(1),
  message: z.enum(['asking_price', 'make_offer', 'check_availability']),
```

```
});

QUICK WINS:
- Use Tailwind UI components (copy/paste)
- Toast notifications for all actions
- Loading states with simple spinners
- Error boundaries on pages
- Mobile responsive from start

DEPLOYMENT (End of Phase 0):
- Push to GitHub
- Deploy to Vercel (free)
- Use Vercel KV for sessions (free tier)
- Upgrade to Postgres later
```

Success Criteria for Phase 0:

- [ ] Users can register and login
- [ ] Users can create listings with ticket uploads
- [ ] Users can make offers on listings
- [ ] Sellers can accept/reject offers
- [ ] Mock payment completes the flow
- [ ] Buyers can "download" their tickets
- [ ] Everything works on mobile
- [ ] Deployed to Vercel

Time Estimate: 2-3 days maximum

Next Steps After Phase 0: Once this prototype works, you can:

1. Migrate to Supabase (Phase 1)
2. Add proper authentication
3. Implement real payments
4. Add all the security features
5. Scale the architecture

The key is proving the concept works before building infrastructure.

PHASE 1: Migration to Production Infrastructure

Objective: Migrate the working prototype from Phase 0 to production-ready infrastructure with Supabase, proper auth, and better architecture.

Instructions for Claude Code:

Now that you have a working prototype, let's upgrade the infrastructure:

MIGRATION STEPS:

1. Set up Supabase project (free tier):
   - Create account at supabase.com
   - Create new project
   - Note connection strings

2. Migrate from SQLite to PostgreSQL:
   - Update prisma/schema.prisma datasource
   - Export existing data if needed
   - Run migrations on Supabase

3. Replace simple auth with Supabase Auth:
   - Remove bcrypt/JWT code
   - Implement Supabase Auth
   - Migrate existing users
   - Add social logins (optional)

4. Move file storage to Supabase Storage:
   - Create buckets for tickets
   - Migrate existing files
   - Implement signed URLs
   - Add file validation

5. Add the monorepo structure (if needed):
   - Only if you have 1000+ users
   - Only if multiple developers
   - Keep it simple for now

6. Implement basic RLS policies:
   - Start with simple policies
   - Test each one thoroughly
   - Add complexity gradually

The key is to migrate incrementally, testing each change before moving on.

```
ticket-marketplace/
├── apps/
│   └── web/          # Next.js 14 app with API routes
├── packages/
│   ├── database/     # Prisma schema and migrations
│   ├── shared/       # Shared types, utils, and Zod schemas
│   └── ui/          # Shared UI components
├── supabase/        # Supabase local config
```

```
├── docker-compose.yml
├── package.json     # Root package.json
├── pnpm-workspace.yaml
└── turbo.json

Setup steps:
1. Initialize pnpm workspace with Turborepo
2. Install Supabase CLI and run: supabase init
3. Start local Supabase: supabase start (includes PostgreSQL, Auth, Storage)
4. Create Next.js app in apps/web with TypeScript, Tailwind, and App Router
5. Set up Prisma in packages/database pointing to local Supabase

Dependencies:
- Root: turbo, @types/node
- apps/web: next, react, @supabase/supabase-js, @supabase/auth-helpers-nextjs
- packages/database: prisma, @prisma/client
- packages/shared: zod
- Dev tools: typescript, eslint, prettier

Prisma schema (packages/database/schema.prisma):
// Define enums for type safety
enum ListingStatus {
  ACTIVE
  INACTIVE
  SOLD
  DELISTED
}

enum OfferStatus {
  PENDING
  ACCEPTED
  REJECTED
  EXPIRED
  COMPLETED
}

enum TransactionStatus {
  PENDING
  PROCESSING
  COMPLETED
  FAILED
  REFUNDED
}

model User {
  id              String   @id @default(uuid())
  email           String   @unique
```

```
  username        String   @unique
  rating          Decimal  @default(5.0)
  isVerified      Boolean  @default(false)
  totalSales      Int      @default(0)
  stripeAccountId String?

  listings        Listing[]
  sentOffers      Offer[]   @relation("BuyerOffers")
  receivedOffers  Offer[]   @relation("SellerOffers")
  purchases       Transaction[] @relation("BuyerTransactions")
  sales           Transaction[] @relation("SellerTransactions")

  @@index([email])
  @@index([username])
}

model Event {
  id        String   @id @default(cuid())
  name      String
  venue     String
  eventDate DateTime
  eventType String
  listings  Listing[]

  @@index([eventDate])
  @@index([eventType])
}

model Listing {
  id          String        @id @default(cuid())
  sellerId    String
  seller      User          @relation(fields: [sellerId], references: [id])
  eventId     String
  event       Event         @relation(fields: [eventId], references: [id])
  title       String
  description String?
  priceInCents Int          // Store as integer to avoid floating point issues
  quantity    Int
  status      ListingStatus @default(ACTIVE)

  tickets     Ticket[]
  offers      Offer[]

  @@index([sellerId])
  @@index([eventId])
  @@index([status])
}
```

```
model Ticket {
  id               String   @id @default(cuid())
  listingId        String
  listing          Listing  @relation(fields: [listingId], references: [id])
  filePath         String   // e.g., 'tickets/user-123/ticket-abc.pdf'
  watermarkPath    String?  // e.g., 'watermarks/user-123/watermark-abc.png'
  originalFileName String
  fileType         String
  fileSize         Int
  verificationCode String   @unique
  isSold           Boolean  @default(false)
  buyerId          String?

  @@index([listingId])
  @@index([verificationCode])
}

model Offer {
  id         String      @id @default(cuid())
  listingId  String
  listing    Listing     @relation(fields: [listingId], references: [id])
  buyerId    String
  buyer      User        @relation("BuyerOffers", fields: [buyerId], references:
[id])
  sellerId   String
  seller     User        @relation("SellerOffers", fields: [sellerId], references:
[id])
  status     OfferStatus @default(PENDING)
  offerData  Json        // {priceInCents, quantity, originalPriceInCents}

  messages   OfferMessage[]
  transaction Transaction?

  @@index([buyerId])
  @@index([sellerId])
  @@index([status])
}

model OfferMessage {
  id             String   @id @default(cuid())
  offerId        String
  offer          Offer    @relation(fields: [offerId], references: [id])
  senderId       String
  templateId     String
  messageData    Json?
  renderedMessage String
```

```
  createdAt      DateTime @default(now())

  @@index([offerId])
}

model Transaction {
  id                    String            @id @default(cuid())
  offerId               String            @unique
  offer                 Offer             @relation(fields: [offerId], references:
[id])
  buyerId               String
  buyer                 User              @relation("BuyerTransactions", fields:
[buyerId], references: [id])
  sellerId              String
  seller                User              @relation("SellerTransactions", fields:
[sellerId], references: [id])
  amountInCents         Int               // Total amount in cents
  platformFeeInCents    Int               // Your commission in cents
  sellerPayoutInCents   Int               // Amount to seller in cents
  status                TransactionStatus @default(PENDING)
  stripePaymentIntentId String?
  stripeTransferId      String?
  completedAt           DateTime?

  @@index([buyerId])
  @@index([sellerId])
  @@index([status])
}

Create these Zod schemas in packages/shared/schemas/:
- userSchema
- listingSchema (with priceInCents validation)
- offerSchema (with enum validation)
- messageTemplateSchema

Set up service abstractions in packages/shared/services/:
- FileStorageService (interface)
- PaymentService (interface)
- EmailService (interface)

Local implementations:
- LocalFileStorage (saves to ./uploads)
- MockPaymentService (simulates Stripe)
- ConsoleEmailService (logs to console)

Set up Supabase RLS policies (critical for security):
-- Enable RLS on all tables
```

```
ALTER TABLE profiles ENABLE ROW LEVEL SECURITY;
ALTER TABLE listings ENABLE ROW LEVEL SECURITY;
ALTER TABLE tickets ENABLE ROW LEVEL SECURITY;
ALTER TABLE offers ENABLE ROW LEVEL SECURITY;
ALTER TABLE transactions ENABLE ROW LEVEL SECURITY;

-- User policies
CREATE POLICY "Users can view all profiles" ON profiles
  FOR SELECT USING (true);

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

-- Listing policies
CREATE POLICY "Anyone can view active listings" ON listings
  FOR SELECT USING (status = 'ACTIVE');

CREATE POLICY "Users can create listings" ON listings
  FOR INSERT WITH CHECK (auth.uid() = seller_id);

CREATE POLICY "Users can update own listings" ON listings
  FOR UPDATE USING (auth.uid() = seller_id);

-- Ticket policies (strict - only buyers can see)
CREATE POLICY "Sellers can view own tickets" ON tickets
  FOR SELECT USING (
    EXISTS (
      SELECT 1 FROM listings
      WHERE listings.id = tickets.listing_id
      AND listings.seller_id = auth.uid()
    )
  );

CREATE POLICY "Buyers can view purchased tickets" ON tickets
  FOR SELECT USING (buyer_id = auth.uid() AND is_sold = true);

-- Continue with policies for offers and transactions...
```

Checkpoint:

- [ ] Monorepo structure created
- [ ] Supabase running locally (database + auth + storage)
- [ ] Prisma schema with proper types (Int for money, Enums for status)
- [ ] RLS policies implemented for all tables
- [ ] Zod schemas created and shared
- [ ] Service interfaces defined

- [ ] Database indexes added for performance
- [ ] Can run everything with single command

PHASE 2: Core API with Service Abstractions

Objective: Build Next.js API routes with proper abstractions, RLS policies, and direct-to-storage uploads.

Instructions for Claude Code:

Create API routes in apps/web/app/api/ with these features:

1. Auth endpoints using Supabase Auth:
   - POST /api/auth/register - Create user with Supabase
   - POST /api/auth/login - Sign in with Supabase
   - POST /api/auth/logout - Sign out
   - GET /api/auth/me - Get current user

2. File handling with signed URLs (efficient pattern):
   - POST /api/tickets/upload-url - Generate signed upload URL
     * Validate user is authenticated
     * Generate unique file path: tickets/{userId}/{ticketId}.pdf
     * Create signed URL for direct upload to Supabase Storage
     * Return URL to client for direct upload
   - POST /api/tickets/process - After upload, process the ticket
     * Generate watermark with Sharp or Supabase transformations
     * Create database record with file paths (not full URLs)
     * Return ticket metadata

3. Listing endpoints with RLS enforcement:
   - GET /api/listings - Paginated, with filters
   - GET /api/listings/[id] - Single listing (hide ticket paths)
   - POST /api/listings - Create with ticket validation
   - PUT /api/listings/[id] - Update own listings
   - DELETE /api/listings/[id] - Soft delete

4. Structured offer system:
   - POST /api/offers - Create with template validation
   - GET /api/offers - Get user's offers
   - POST /api/offers/[id]/respond - Template responses only
   - GET /api/offers/[id]/messages - Get conversation

5. Mock payment flow:
   - POST /api/payments/create-intent - Mock Stripe intent
   - POST /api/payments/confirm - Simulate payment
   - Automatic ticket delivery on success
   - Update transaction records

Implement these utilities:
- withAuth() middleware using Supabase session
- withValidation() using Zod schemas
- withRateLimit() for API protection
- withErrorHandler() for consistent errors

Service implementations:
- SupabaseStorageService:

```
  * generateUploadUrl(path, options)
  * getSignedDownloadUrl(path, expiresIn)
  * deleteFile(path)
- SharpImageProcessor for watermarks (or use Supabase image transformations)
- MockStripeService for payments
- PrismaClient with RLS context

Critical: Write corresponding RLS policies as you build each endpoint:
- When building listing creation, add the INSERT policy
- When building offer responses, add the UPDATE policy
- Test that RLS blocks unauthorized access

Environment variables:
- NEXT_PUBLIC_SUPABASE_URL (from supabase status)
- NEXT_PUBLIC_SUPABASE_ANON_KEY (from supabase status)
- SUPABASE_SERVICE_ROLE_KEY (for backend only)
- MOCK_PAYMENTS=true
- PLATFORM_FEE_PERCENT=6
```

Checkpoint:

- [ ] All API routes working locally
- [ ] Signed URL uploads working (no server bandwidth used)
- [ ] RLS policies blocking unauthorized access
- [ ] File paths stored, not full URLs
- [ ] Service abstractions in place
- [ ] Mock payments flowing correctly
- [ ] Database queries optimized with Prisma

PHASE 3: Frontend with Caching & Validation

Objective: Build the UI with proper state management, form validation, and caching for optimal free-tier usage.

Instructions for Claude Code:

Build the frontend in apps/web using:

1. Set up providers (app/layout.tsx):
   - Supabase Auth Provider
   - TanStack Query Provider
   - Theme Provider (dark mode ready)
   - Toast notifications

2. Install and configure:
   - @tanstack/react-query for API calls
   - react-hook-form with Zod resolver
   - shadcn/ui components (Button, Card, Dialog, etc.)
   - lucide-react for icons

3. Authentication flow:
   - /auth/login - Supabase email/password
   - /auth/register - With username
   - /auth/forgot-password
   - Protect routes with middleware

4. Main pages:
   - / - Homepage with search
   - /events - Browse events
   - /listings - Browse tickets
   - /listings/[id] - Detail page
   - /listings/create - Multi-step form

5. Ticket upload component:
   - Drag and drop with react-dropzone
   - Client-side validation (PDF/image only)
   - Progress indicator
   - Preview uploaded tickets
   - Show watermarked versions

6. Offer system UI:
   - Structured message templates only
   - No free text inputs anywhere
   - Clear offer flow visualization
   - Status badges and timelines

7. Dashboard pages:
   - /dashboard - Overview
   - /dashboard/listings - Manage listings
   - /dashboard/offers - Sent/received
   - /dashboard/purchases - Bought tickets
   - /dashboard/sales - Sold tickets

```
8. Implement granular caching strategy:
   - Listings page: 1-2 minute cache (changes frequently)
   - Individual listing: 10 minute cache (changes less)
   - User profile: Cache for session duration
   - Use stale-while-revalidate for instant UI
   - Prefetch next page of results
   - Invalidate specific queries on mutations

9. Form schemas (reuse from packages/shared):
   - CreateListingSchema
   - MakeOfferSchema
   - All forms use react-hook-form + Zod

10. Optimistic updates for better UX:
    - When sending offer: Update UI immediately
    - When responding: Show response instantly
    - On error: Rollback and show error toast
    - TanStack Query handles rollback automatically

11. Mock payment UI:
    - /payment/[offerId] - Fake Stripe Elements
    - Success/failure handling
    - Automatic redirect after payment
```

Checkpoint:

- [ ] Auth flow working with Supabase
- [ ] File uploads with progress
- [ ] Forms validated with Zod
- [ ] Caching reducing API calls
- [ ] Responsive on mobile
- [ ] Mock payments complete flow

PHASE 4: MVP Features & Polish

Objective: Complete core features and prepare for real users, keeping scope minimal.

Instructions for Claude Code:

Complete the MVP with these focused features:

1. Ticket delivery system:
   - Automatic delivery after payment
   - Secure download URLs (time-limited)
   - Email delivery with SendGrid later
   - Download tracking in database

2. Basic fraud prevention:
   - Duplicate ticket detection (file hash comparison)
   - Rate limiting on downloads
   - Verification code display
   - Report suspicious listings
   - Future enhancement: OCR-based duplicate detection
     (extract seat numbers, barcodes for deeper validation)

3. Ticket delivery with signed URLs:
   - Use Supabase createSignedUrl() for secure downloads
   - 5-minute expiration for download links
   - Track download attempts in database
   - Email backup delivery with time-limited link
   - Construct URLs from stored paths, not database URLs

4. Seller verification:
   - Basic badge after 5 successful sales
   - Show completion rate
   - Display member since date
   - Average rating calculation

5. Search and filters:
   - Event name autocomplete
   - Date range picker
   - Price range slider
   - Sort by date/price
   - "Verified seller" filter

6. SEO and performance:
   - generateMetadata() for all pages
   - OpenGraph images
   - Sitemap.xml
   - Loading.tsx states
   - Error boundaries

7. Legal pages (markdown):
   - /terms - Basic terms
   - /privacy - GDPR compliant

```
   - /refunds - Clear policy
   - Cookie consent banner

8. Trust features:
   - "How it works" page
   - Trust badges on homepage
   - Sample watermark display
   - Security features list

9. Essential monitoring:
   - Vercel Analytics (free)
   - Error boundary reporting
   - API response logging
   - User behavior events

Do NOT build yet:
- Admin panels
- Bulk operations
- Complex analytics
- Email campaigns
- A/B testing
- Forums/chat
```

Checkpoint:

- [ ] Ticket delivery working
- [ ] Basic fraud prevention active
- [ ] Search and filters functional
- [ ] Legal pages complete
- [ ] SEO optimized
- [ ] Ready for beta users

PHASE 5: Payment Integration & Production Prep

Objective: Integrate Stripe Connect and prepare for production deployment on free tiers.

Instructions for Claude Code:

Prepare for real money handling:

1. Stripe Connect setup:
   - Create Stripe account (free)
   - Choose Connect Standard (no monthly fee)
   - Implement OAuth flow for sellers
   - Store stripe_account_id in database

2. Update payment service with critical webhook security:
   - Real StripeService implementation
   - Payment intents with metadata
   - Application fee (your commission)
   - CRITICAL: Webhook endpoint is source of truth
     * Frontend redirects are ONLY for UX
     * ALL business logic in webhook handler:
       - Update transaction status
       - Mark tickets as sold
       - Trigger ticket delivery
       - Update seller balance
     * Implement Stripe signature verification
     * Idempotency handling for duplicate events
   - Automatic transfers to sellers via Connect

3. Environment configuration:
   - Create .env.production
   - Stripe keys (test mode first)
   - MOCK_PAYMENTS=false
   - Update service factories

4. Security hardening:
   - Implement CSP headers
   - Add rate limiting (upstash free tier)
   - Validate all inputs
   - Sanitize file uploads
   - CORS configuration

5. Update Supabase project:
   - Create cloud project (free tier)
   - Apply migrations
   - Set up RLS policies
   - Configure storage buckets
   - Enable email auth

6. Deployment preparation:
   - Optimize images with next/image
   - Enable SWC minification

```
   - Set up error tracking
   - Configure redirects
   - Test build locally

7. Testing checklist:
   - Full purchase flow
   - Seller onboarding
   - Payment processing
   - Ticket delivery
   - Refund handling
   - Mobile testing
```

Checkpoint:

- [ ] Stripe Connect integrated
- [ ] Seller onboarding working
- [ ] Test payments processing
- [ ] Security measures active
- [ ] Cloud Supabase ready
- [ ] Builds passing

PHASE 6: Deploy & Launch Strategy

Objective: Deploy to production and launch with zero monthly costs until revenue.

Instructions for Claude Code:

Deploy and launch systematically:

1. Vercel deployment:
   - Connect GitHub repo
   - Configure environment variables
   - Set up preview deployments
   - Custom domain (when ready)
   - Enable Analytics

2. Supabase configuration:
   - Enable RLS on all tables
   - Set up email templates
   - Configure rate limits
   - Enable captcha
   - Set spending caps

3. Stripe configuration:
   - Switch to live keys gradually
   - Set up webhook endpoint
   - Configure payout schedule
   - Enable fraud tools
   - Test dispute handling

4. Monitoring setup:
   - Vercel Analytics (free)
   - Supabase Dashboard
   - Stripe Dashboard
   - UptimeRobot (free)
   - Google Search Console

5. Soft launch plan:
   - Start with test mode
   - Invite beta users
   - Monitor all metrics
   - Fix critical issues
   - Enable live payments

6. Performance optimization before scaling:
   - FIRST: Add database indexes (free!)
     * Already included in Prisma schema
     * Monitor slow queries in Supabase dashboard
     * Add composite indexes if needed
   - Enable gzip compression
   - Implement caching headers
   - Optimize images
   - Minify assets

```
   - Enable CDN (Vercel's included)

7. When to upgrade (based on revenue):
   - Always try indexes before upgrading database
   - Vercel Pro: >100GB bandwidth
   - Supabase Pro: >500MB database or slow queries
   - Custom domain: Immediately ($10-15/year)
   - Email service: >3K emails/month
   - Monitor usage daily to avoid surprises
```

Checkpoint:

- [ ] Deployed to Vercel
- [ ] Supabase production ready
- [ ] Stripe test mode working
- [ ] Monitoring active
- [ ] Beta users testing
- [ ] Costs still under $20/month

Cost Timeline

Development (Months 1-2): $0

- Everything runs locally with Supabase CLI
- No external services needed

Beta Testing (Month 3): $0-10

- Free tiers only
- Custom domain ($10)

Soft Launch (Months 4-5): $10-20

- Domain + minor overages
- Still on free tiers

Growth Phase (Month 6+): Scale with revenue

- Upgrade only when revenue justifies
- 6% commission covers costs
- Reinvest profits

Key Improvements Made

1. **Supabase CLI from start** - No migration pain

2. **Monorepo structure** - Shared types, one deployment
3. **Prisma ORM** - Type-safe, AI-friendly
4. **Service abstractions** - Swap implementations easily
5. **Stripe Connect** - Proper marketplace payments
6. **TanStack Query** - Reduces API calls
7. **Zod validation** - Shared schemas
8. **Sharp for images** - Local watermarking
9. **Focused MVP** - Removed unnecessary features
10. **Clear upgrade triggers** - Know when to spend

Critical Security & Performance Updates

Based on expert review, these crucial improvements were added:

1. **Money as Integers** - Store cents, not dollars (no float errors)
2. **Enums for Status** - Type-safe status fields prevent bugs
3. **File Paths, Not URLs** - Store paths in DB, generate URLs at runtime
4. **RLS from Day One** - Security policies as you build, not after
5. **Signed URL Uploads** - Direct to storage, bypass server bandwidth
6. **Webhook Source of Truth** - Payment confirmation only via webhooks
7. **Database Indexes First** - Free performance boost before paid upgrades
8. **Optimistic Updates** - Instant UI feedback for better UX
9. **Granular Caching** - Different cache times for different data

Success Metrics

- Customer Acquisition Cost < $5
- Average Transaction Value > $50
- Platform Take Rate: 6%
- Ticket Delivery Success: >99%
- Seller Verification Rate: >80%
- Dispute Rate: <1%

Total Timeline

- Phase 1: 2-3 days (Monorepo + Supabase)
- Phase 2: 3-4 days (API + Services)
- Phase 3: 4-5 days (Frontend + Auth)

- Phase 4: 3-4 days (MVP Features)
- Phase 5: 3-4 days (Stripe + Security)
- Phase 6: 2-3 days (Deploy + Launch)

Total: 17-25 days (more realistic than original estimate)

Monthly Cost: $0-20 until significant revenue