Project Handover Report- Clothing Store E-commerce Platform

1. Technical Architecture Overview

Technology Stack

Frontend: React 19 with React Router 7, Progressive Web App (PWA) with service workers for offline capability, localStorage for cart persistence, and Context API for state management (Authentication and Shopping Cart).

Backend: Node.js with Express 5, TypeScript for type safety, Passport.js for Google OAuth 2.0 authentication, Express Session with MongoDB session store for session management.

Database: MongoDB with Mongoose ODM, containing collections for Users, Products, Categories, Sizes, Stock (inventory), Orders, and saved payment Cards.

Deployment: Currently configured for local development (Frontend: localhost:3001, Backend: localhost:3000). Production deployment in progress. Will deploy on render (backend ok at https://clothing-store-p73x.onrender.com/)

System Architecture & Data Flow

The application follows a standard client-server architecture with the following data flow:

1. **User Authentication:** User initiates Google OAuth login → Backend redirects to Google → Google callback returns to backend → Session created in MongoDB → Frontend receives session token → User state stored in React Context
2. **Product Browsing:** Frontend requests products by category → Backend queries MongoDB with category filter → Products with populated category references returned → Images served from static folder
3. **Shopping Cart:** User adds items → Cart stored in browser localStorage → On checkout, cart data sent to backend → Backend validates stock availability → Stock quantities decremented → Order created in MongoDB
4. **Order Management:** Orders linked to users via userId reference → Admin panel fetches all orders with populated user and product data → Status updates trigger database updates

Key Design Decisions

Google OAuth over custom authentication: Reduces security risks by delegating authentication to Google, eliminates need for password management, and provides trusted user verification. This was chosen to accelerate development and improve security posture.

MongoDB over SQL: Selected for flexible schema design allowing easy addition of product attributes, natural JSON structure matching frontend/backend data flow, and simpler deployment without complex migrations. The product catalog's semi-structured nature fits NoSQL better than rigid relational tables.
PWA Implementation: Enables offline browsing of cached products, provides app-like experience with install prompts, and improves loading performance through service worker caching. This enhances user experience especially on mobile devices and slow networks.
Context API over Redux: Sufficient for current application scale, reduces boilerplate code, and provides simpler learning curve. The application has only two global states (auth and cart), making Redux's complexity unnecessary.
Separate Size and Stock Models: Allows flexible inventory management where same product can have different stock levels per size and color combination. This supports realistic e-commerce scenarios where availability varies by variant.

2. Technical Debt & Known Limitations

Current Problems

Security Issues:

- Payment card data stored in plain text including full card numbers (CVV stored temporarily). This violates PCI DSS compliance and exposes sensitive data. Real applications must use payment processors like Stripe.
- Session secret hardcoded in .env file should be rotated regularly and stored in secure key management system.
- No rate limiting on API endpoints allows potential DDoS attacks.
- CORS configured for specific localhost URLs only; needs environment-based configuration for production.

Code Quality Issues:

- Mixed TypeScript (.ts) and JavaScript (.js) files create inconsistent type safety. Controllers are in JavaScript while routes/models are TypeScript.
- Inconsistent error handling- some routes have try-catch blocks, others don't. Missing centralized error handling middleware.
- No input validation on most endpoints (missing express-validator or joi). User input directly passed to database queries.
- Test files present but not implemented (App.test.js contains only placeholder test).

Performance Limitations:

- All product images loaded on initial page load without lazy loading or pagination.
- No caching strategy for frequently accessed data (categories, sizes).

- Database queries not optimized- missing indexes on frequently queried fields like productId, userId.
- Stock updates not transaction-safe; concurrent orders could oversell inventory.

Incomplete Features:

- Admin panel has "Edit" and "Delete" buttons on products but handlers not implemented.
- Order tracking system non-functional- status updates don't trigger notifications.
- Email confirmation for orders not implemented.
- Product search/filter functionality missing (only category filtering exists).
- User profile editing not possible (name, avatar cannot be changed).
- No password reset flow (though using OAuth reduces this need).

Deployment Gaps:

- No environment configuration for production builds.
- Database seeding script (seed.js) contains hardcoded data with no proper production data strategy.
- Static image files served from local filesystem with hardcoded localhost URLs in database.
- No CI/CD pipeline or deployment documentation.
- Missing health check endpoints for monitoring.

Why These Are Problematic

The plain text card storage is the most critical issue - a data breach would expose complete payment information causing legal liability, financial losses, and destroyed customer trust. The lack of input validation creates SQL/NoSQL injection vulnerabilities allowing attackers to manipulate or delete data. Missing transaction handling means race conditions during checkout can cause inventory inconsistencies, overselling products. The hardcoded localhost image URLs will break entirely in production, preventing any product images from displaying.

Intentional Prioritization Decisions

Payment processing: Full Stripe/PayPal integration deprioritized to focus on core e-commerce flow. Current implementation serves as UI prototype but must not go to production.

Email notifications: Skipped to meet deadline - requires email service integration (SendGrid, AWS SES) which adds infrastructure complexity.

Advanced product filtering: Basic category filtering implemented; comprehensive search with size/color/price filters deemed non-critical for MVP.

Automated testing: Test infrastructure present but not implemented due to time constraints. Manual testing performed instead.
Image upload system: Products use external URLs; admin upload interface postponed. Current approach sufficient for demo but limits content management.

3. Future Development Recommendations

Must Implement (Security & Critical Functionality)

Integrate proper payment processor: Replace card storage with Stripe Elements or PayPal SDK. Store only payment method tokens, never raw card data. This is non-negotiable for production launch.
Add comprehensive input validation: Implement express-validator on all POST/PATCH endpoints. Validate data types, required fields, string lengths, and sanitize inputs. Prevents injection attacks and data corruption.
Implement proper authentication middleware: Add JWT refresh tokens alongside sessions. Create proper admin authorization middleware checking isAdmin flag before sensitive operations. Currently admin routes rely on trust.
Add database transactions for orders: Wrap order creation and stock updates in MongoDB transactions. Ensures atomicity - either complete order succeeds or entire operation rolls back, preventing inventory inconsistencies.
Configure production environment: Create separate .env files for development/staging/production. Use environment variables for all URLs, implement proper CORS configuration, add rate limiting middleware, and set up health check endpoints.

Should Implement (Enhanced Functionality)

Order email notifications: Integrate SendGrid/AWS SES to send confirmation emails with order details and tracking information. Improves customer experience and reduces support inquiries.
Product search and advanced filtering: Add full-text search on product names/descriptions using MongoDB Atlas Search or Elasticsearch. Implement filters for price range, sizes, colors, and sorting options.
Image management system: Build admin interface for uploading product images to cloud storage (AWS S3, Cloudinary). Store CDN URLs in database instead of localhost paths.
Inventory alerts: Add low-stock notifications for admin dashboard. Implement automatic reorder suggestions when inventory falls below threshold.
Order tracking: Create detailed order status workflow (pending → processing → shipped → delivered) with automatic status updates and customer-facing tracking page.

Testing suite: Implement unit tests for controllers, integration tests for API routes, and E2E tests for critical flows (checkout, authentication). Target 70%+ code coverage.

Should Improve (Code Quality & Performance)

Complete TypeScript migration: Convert all JavaScript controllers to TypeScript. Add proper type definitions for request/response objects. Enable strict mode in tsconfig.json for better type safety.

Add database indexes: Create indexes on userId, productId, categoryId, and order status fields. Implement compound indexes for frequently combined queries (productId + sizeId + color for stock lookups).

Implement caching layer: Add Redis for caching product lists, categories, and sizes. Set appropriate TTL values. Reduces database load and improves response times significantly.

Centralized error handling: Create global error handler middleware catching all errors. Implement proper error classes (ValidationError, AuthError, NotFoundError). Return consistent error response format.

API versioning strategy: Current routes use /api/v1/ prefix but no versioning strategy defined. Document approach for introducing breaking changes without disrupting existing clients.

Refactor large components: Break down ProductsPage, Checkout, and AdminPage components into smaller, reusable pieces. Extract custom hooks for common logic (useAuth, useCart already exist but could be expanded).

Optimize bundle size: Implement code splitting on routes. Lazy load admin components only when needed. Use React.lazy() and Suspense for performance improvements.

Documentation: Add JSDoc comments to complex functions. Create API documentation using Swagger/OpenAPI. Document deployment procedures and environment setup in detail.

Recommended Priority Order: Security fixes → Production deployment → Testing → Performance optimization → Feature enhancements

Estimated Timeline: Critical security fixes (2 weeks) → Production deployment setup (1 week) → Core functionality completion (3 weeks) → Performance optimization (2 weeks) → Advanced features (4+ weeks)