Darksword Armory Clone — Agent Replication Guide

Author: Manus AI
Date: February 14, 2026
Purpose: This document provides a complete guide for an AI agent to replicate, deploy, and test the Darksword Armory e-commerce clone. It covers the full project architecture, data pipeline, design decisions, and step-by-step instructions for Neon PostgreSQL deployment on Vercel.

Table of Contents

1. Project Overview
2. Design Specifications
3. Data Sources and Pipeline
4. Database Schema Architecture
5. Product Variation and Attribute System
6. Key Technical Decisions
7. Neon PostgreSQL Setup and Testing
8. Vercel Deployment Guide
9. Testing Checklist
10. Known Issues and Future Work

---

1. Project Overview

This project is a full-stack e-commerce clone of darksword-armory.com, a Canadian company that hand-forges medieval swords, armor, and weapons. The design

template is based on [davidprotein.com](), which uses a dark editorial luxury aesthetic with gold accents.

The stack consists of React 19 with TypeScript on the frontend, Express 4 with tRPC 11 on the backend, Drizzle ORM for database access, and Tailwind CSS 4 for styling. The project was originally built with a MySQL/TiDB database and has been converted to PostgreSQL for Neon deployment.

Component	Technology	Notes
Frontend Framework	React 19 + TypeScript	Vite 6 build tooling
Routing	Wouter 3	Lightweight client-side router
Styling	Tailwind CSS 4 + shadcn/ui	Dark theme with OKLCH color format
API Layer	tRPC 11 + Superjson	End-to-end type safety
Backend	Express 4	Serves both API and static assets
ORM	Drizzle ORM 0.44	PostgreSQL dialect (`pg-core`)
Database Driver	`pg` (node-postgres)	For standard PostgreSQL; swap to `@neondatabase/serverless` for Neon
Auth	Manus OAuth	Session cookies + JWT
Testing	Vitest 2	24 tests across 4 test files

---

2. Design Specifications

The visual design replicates the davidprotein.com layout while using Darksword Armory's brand identity. Below are the exact design tokens used throughout the application.

Color Palette

Token	Value	Usage
Background	`#0A0A0A`	Page background
Foreground	`#F5F0E8`	Primary text
Gold Accent	`#C8A45C`	CTAs, hover states, badges, links
Gold Hover	`#D4B06A`	Button hover states
Card Background	`#111111`	Product cards, sections
Muted	`#1A1A1A`	Secondary backgrounds
Border	`#2A2A2A`	Dividers, card borders

Typography

Font	Usage	Source
Cinzel Decorative	Logo, hero headings	Google Fonts CDN
Cormorant Garamond	Section headings, product titles	Google Fonts CDN
DM Sans	Body text, UI elements	Google Fonts CDN

Layout Structure

The homepage follows this exact section order: Announcement Bar (rotating messages), Sticky Header with navigation, Hero Section (video with auto-transition to image carousel), Scrolling Marquee Trust Bar, Product Carousel (featured products), Craftsman/Founder Section (Eyal Azerad), Tabbed Specifications Section, Brand Story Section, Secondary Marquee Bar, Bundle Promotion Section, Category Showcase Grid, Customer Reviews Carousel, and Footer with newsletter signup.

---

3. Data Sources and Pipeline

WooCommerce XML Exports

All product data originates from 31 WordPress/WooCommerce XML export files located in the `data/xml/` directory. These files contain the complete product catalog, variations, attributes, categories, media references, blog posts, and static pages.

The data extraction pipeline works as follows. First, the XML files are parsed to extract product records including title, slug, content, excerpt, price, sale price, SKU, stock status, thumbnail URL, gallery URLs, categories, and custom meta fields. Second, product variations are extracted from the XML — these include Package options (e.g., "Blunt blade & Scabbard"), Size, Model, Color, and Options attributes with their individual prices. Third, product attributes (non-price-affecting customization options) are extracted from WooCommerce taxonomy terms: `pa_grip` (6 color swatches), `pa_scabbard` (1 color swatch), `pa_blade-finish` (dropdown), and `pa_guard-pommel-finish` (dropdown).

Extracted Data Files

File	Records	Description
`data/products.json`	367	Full product catalog with metadata
`data/variations.json`	1,150	Product variations with prices
`data/media.json`	—	Media library references
`data/posts.json`	—	Blog posts
`data/pages.json`	—	Static pages (About, FAQ, etc.)

Data Extraction Script

The extraction script (`extract-data.mjs`) reads all XML files, parses them with a streaming XML parser, and outputs structured JSON. If the extracted data files are missing, the agent should re-run the extraction from the XML sources. The XML files in `data/xml/` are the authoritative source of truth.

---

4. Database Schema Architecture

The PostgreSQL schema contains 12 tables organized into four functional groups.

Core Commerce Tables

The **products** table stores 367 items with fields for name, slug, description, short description, price (decimal 10,2), sale price, SKU, stock quantity, stock status (enum: instock/outofstock/onbackorder), image URL, gallery images (jsonb array), category ID (foreign key), categories (jsonb array of category names for fast filtering), physical specs (weight, steel type, blade length, etc.), featured flag, and status (enum: publish/draft/trash).

The **categories** table holds 62 categories with name, slug, description, image URL, parent ID for hierarchy, and sort order.

The **product_variations** table contains 1,150 records linking to products via `product_id`. Each variation has a variation type (package/size/model/color/options), option name, option value, absolute price, regular price, sale price, price modifier, SKU, image URL, stock status, and a WordPress variation ID for traceability.

The **product_attributes** table stores 503 records of non-price-affecting customization options. Each record has an attribute key (e.g., “grip”, “blade-finish”), display name, display type (enum: color_swatch/dropdown), and an options jsonb array containing value strings with optional color hex codes and default flags.

Order Management Tables

The **orders** table tracks order number (unique), user ID, status (7-value enum), payment status (5-value enum), payment method, transaction ID, subtotal/shipping/tax/discount/total amounts, customer contact info, shipping and billing addresses (jsonb), notes, tracking number, and shipping carrier.

The **order_items** table links to orders and stores product ID, variation ID, product name snapshot, SKU, quantity, unit price, line total, and variation details (jsonb).

Cart and User Tables

The **carts** table supports both authenticated (user ID) and guest (session ID) shopping. The **cart_items** table stores product ID, variation ID, quantity, price, and variation details. The **users** table backs the Manus OAuth flow with open ID, name, email, login method, role (user/admin), and timestamps. Additional tables include **addresses**, **wishlist_items**, and **reviews**.

---

5. Product Variation and Attribute System

This is the most complex part of the data model and deserves special attention.

Variations (Price-Affecting)

Variations come from WooCommerce's variable product system. A single product like "The Medieval Bastard Sword" may have 4-6 Package variations, each with a distinct absolute price. For example: "Blunt blade & Scabbard" at 580.00, "Sharp pointed blade & Scabbard" at 605.00, "Sharp pointed blade, Scabbard & Sword belt" at $735.00. The frontend displays these as a dropdown selector, and selecting a variation updates the displayed price.

The variation type field categorizes variations into Package (most common for swords), Size, Model, Color, and Options. The `option_name` field holds the attribute name (e.g., "Package") and `option_value` holds the specific choice text.

Attributes (Non-Price-Affecting)

Attributes are cosmetic customization options that do not change the price. They are displayed on the product page as either color swatches or dropdown selectors.

Attribute	Display Type	Options
Grip	Color Swatch	Black, Brown, Dark Brown, Burgundy, Red, Tan (with hex codes)
Scabbard	Color Swatch	Black (with hex code)
Blade (Finish)	Dropdown	High polish, Satin finish
Guard & Pommel (Finish)	Dropdown	Highly polished, Satin Finish

The grip color swatches use these exact hex values: Black (#1A1A1A), Brown (#6B3A2A), Dark Brown (#3B1E0E), Burgundy (#5C1A1A), Red (#8B1A1A), Tan (#C4A46C).

---

6. Key Technical Decisions

Product Card Hover Effect (React State vs CSS)

Tailwind CSS 4 wraps `group-hover:` utilities inside `@media (hover: hover)`, which does not trigger in all browser environments. The product card hover effects (image swap to second gallery image, zoom scale 1.1x, gold title color change) are implemented using React `onMouseEnter` / `onMouseLeave` state handlers with inline styles, bypassing the Tailwind limitation entirely. This is documented in `ProductCard.tsx`.

Category Slug Matching (Resilient Fallback)

WooCommerce uses long compound slugs like `samurai-swords-katanas-japanese-swords` while the frontend navigation uses short slugs like `samurai-swords`. The `getCategoryBySlug` function in `server/db.ts` implements a four-tier fallback: exact match, starts-with match, contains match, and name-based ILIKE match. This ensures all navigation links resolve correctly.

Price Display for Variable Products

Variable products display a price range (e.g., "USD605.00–USD735.00") computed from the MIN and MAX variation prices. The `enrichWithVariationInfo` helper in `server/db.ts` fetches variation count and price range in a single aggregation query and attaches it to product listings. Products with zero-price variations are excluded from the MIN/MAX calculation using a `CASE WHEN price > 0` filter.

JSON Storage in PostgreSQL

Category names are stored as a `jsonb` array in the products table (in addition to the `category_id` foreign key) to enable fast text-based filtering without joins. Gallery images are also stored as a `jsonb` array of URL strings. The `categories` field is queried using `::text ILIKE '%name%'` for category-based product filtering.

---

7. Neon PostgreSQL Setup and Testing

The codebase has been fully converted from MySQL to PostgreSQL. The following steps describe how to set up and test with a Neon database.

Step 1: Create a Neon Database

1. Go to neon.tech and create a free account
2. Create a new project (e.g., "darksword-armory")
3. Copy the connection string — it will look like:
   `postgresql://username:password@ep-xxxxx.us-east-2.aws.neon.tech/neondb?sslmode=require`

Step 2: Configure Environment

Set the `DATABASE_URL` environment variable to the Neon connection string. If deploying on Vercel, add it in the Vercel dashboard under Settings → Environment Variables.

For local development, create a `.env` file:

```
DATABASE_URL=postgresql://username:password@ep-xxxxx.us-east-
2.aws.neon.tech/neondb?sslmode=require
```

Step 3: Optional — Switch to Neon Serverless Driver

The current codebase uses the standard `pg` (node-postgres) driver, which works with any PostgreSQL database including Neon. However, for optimal performance on Vercel's serverless functions, you may want to switch to the `@neondatabase/serverless` driver.

To do this, install the Neon driver (`pnpm add @neondatabase/serverless`), then update the `getDb()` function in `server/db.ts`:

```
import { drizzle } from "drizzle-orm/neon-http";
import { neon } from "@neondatabase/serverless";

export async function getDb() {
  if (!_db && process.env.DATABASE_URL) {
    const sql = neon(process.env.DATABASE_URL);
    _db = drizzle(sql);
  }
  return _db;
}
```

Similarly, update the seed script to use `@neondatabase/serverless` instead of `pg`.

Step 4: Push Schema

Run the migration command to create all tables:

```
pnpm db:push
```

This generates a SQL migration file and applies it. You should see "12 tables" in the output and "migrations applied successfully" at the end.

Step 5: Seed the Database

Ensure the extracted data files exist at the paths referenced in `seed-db.mjs` (default: `/home/ubuntu/extracted_data/products.json`). If they are missing, copy them from `data/products.json` or re-extract from the XML files.

Then run:

```
node seed-db.mjs
```

Expected output: 62 categories inserted, 367 products inserted, and product variations added for sword products.

Important: The basic seed script adds default variations for sword products. For the full 1,150 variations from the original WooCommerce data, you need to also run the variation seeding logic that reads from `data/variations.json` and the attribute seeding that reads from the XML files. These were originally run as separate scripts during development.

Step 6: Verify Data Integrity

After seeding, verify the data by running these SQL queries against your Neon database:

```
SELECT COUNT(*) FROM products;           -- Expected: 367
SELECT COUNT(*) FROM categories;          -- Expected: 62
SELECT COUNT(*) FROM product_variations;  -- Expected: 1,150+
SELECT COUNT(*) FROM product_attributes;  -- Expected: 503
```

Step 7: Run Tests

```
pnpm test
```

All 24 tests should pass. The test files are:

- `server/auth.logout.test.ts` — Authentication logout flow

- `server/ecommerce.test.ts` — Product listing, category listing, cart operations
- `server/products.test.ts` — Product queries, variations, featured products, related products
- `server/product-card.test.ts` — Product card data enrichment (variation counts, price ranges)

If any tests fail due to PostgreSQL syntax differences, check for:

1. String comparison operators — PostgreSQL uses `ILIKE` for case-insensitive matching (already converted)
2. Type casting — PostgreSQL requires explicit casts like `::numeric` or `::text` (already converted)
3. JSON querying — PostgreSQL uses `jsonb` type and `::text` cast for LIKE operations (already converted)
4. Insert returning — PostgreSQL uses `.returning()` instead of MySQL's `insertId` (already converted)
5. Upsert syntax — PostgreSQL uses `ON CONFLICT DO UPDATE` instead of MySQL's `ON DUPLICATE KEY UPDATE` (already converted)

Step 8: Test the Application

Start the dev server (`pnpm dev`) and verify these critical flows:

1. **Homepage loads** — Hero video plays, product carousel shows products with images and prices
2. **Shop page** — Products display with correct price ranges, category sidebar shows counts
3. **Category pages** — All navigation links resolve (Medieval Swords, Armors, Daggers, Axes, Samurai Swords, Sale)
4. **Product detail page** — Variation selector works, attributes (grip swatches, blade finish dropdown) display correctly
5. **Cart** — Add to cart works, cart drawer opens, quantity updates work
6. **Search** — Returns relevant results

---

8. Vercel Deployment Guide

Build Configuration

The project builds with `pnpm build`, which runs Vite for the frontend and esbuild for the backend, outputting to the `dist/` directory. The entry point is `dist/index.js`.

Create a `vercel.json` in the project root:

```
{
  "version": 2,
  "builds": [
    {
      "src": "dist/index.js",
      "use": "@vercel/node"
    }
  ],
  "routes": [
    {
      "src": "/(.*)",
      "dest": "dist/index.js"
    }
  ]
}
```

Required Environment Variables

Variable	Required	Description
`DATABASE_URL`	Yes	Neon PostgreSQL connection string
`JWT_SECRET`	Yes	Session cookie signing secret (generate a random 64-char string)
`VITE_APP_ID`	Yes	Manus OAuth application ID
`OAUTH_SERVER_URL`	Yes	Manus OAuth backend URL
`VITE_OAUTH_PORTAL_URL`	Yes	Manus login portal URL
`OWNER_OPEN_ID`	Yes	Owner's Manus open ID
`OWNER_NAME`	Yes	Owner's display name
`VITE_APP_TITLE`	No	Site title (default: "Darksword Armory")
`VITE_APP_LOGO`	No	Logo path (default: "/darksword-logo.svg")

---

9. Testing Checklist

The following checklist should be completed by the next agent after deploying to Neon/Vercel.

Database Tests

- [ ] Schema pushes without errors (`pnpm db:push`)
- [ ] Seed script completes without errors (`node seed-db.mjs`)
- [ ] Product count matches expected (367)
- [ ] Category count matches expected (62)
- [ ] Variation count matches expected (1,150+)
- [ ] Attribute count matches expected (503)

Unit Tests

- [ ] All 24 vitest tests pass (`pnpm test`)
- [ ] No TypeScript errors (`pnpm check`)

Functional Tests

- [ ] Homepage loads with hero video and product carousel
- [ ] Shop page displays products with correct prices and images
- [ ] All 7 main category navigation links work (Medieval Swords, Armors, Daggers, Axes, Samurai Swords, Jewelry, Sale)
- [ ] Product detail page shows variation selector with correct prices
- [ ] Product attributes (grip swatches, blade finish) display correctly
- [ ] Add to cart works for both simple and variable products
- [ ] Cart drawer opens and shows correct items/totals
- [ ] Search returns relevant results
- [ ] Checkout form submits and creates an order
- [ ] Admin dashboard loads for admin users

Performance Tests

- [ ] Page load time under 3 seconds
- [ ] Product images load from darksword-armory.com CDN
- [ ] No console errors in browser

---

10. Known Issues and Future Work

Payment Integration (Pending)

The checkout flow is built but payment processing is not yet integrated. The client uses **Elavon Lightbox** as their payment gateway. Integration requires Elavon merchant credentials (Merchant ID, User ID, SSL PIN). The Lightbox JavaScript SDK

should be loaded in the checkout page and the payment form should submit to Elavon's hosted payment page.

QuickBooks Integration (Pending)

Automatic invoice creation in QuickBooks is planned but not yet implemented. This requires QuickBooks API credentials (Client ID, Client Secret, Realm ID). The integration should create an invoice in QuickBooks whenever an order is marked as "paid".

Image Hosting

Product images currently reference URLs on `darksword-armory.com`. For production, these should be migrated to a CDN (e.g., Cloudflare R2, AWS S3) to avoid dependency on the original site's availability.

SEO and Meta Tags

The site does not yet have proper meta tags, Open Graph tags, or structured data (JSON-LD) for product pages. Adding these would improve search engine visibility.

Mobile Optimization

The site is responsive but some components (particularly the product detail page variation selectors) could benefit from additional mobile-specific styling and touch interaction improvements.