PROJECT REPORT

ON

"ArtCraft – Art and Craft Supplies E-Commerce Platform"

IN

"Department of Computer Science and Engineering"

SUBMITTED IN PARTIAL FULFILLMENT OF THE DEGREE

OF

BE(CSE)

Under the Guidance of:

Mr. Sandeep and Mrs. Rakhi
Department of CSE

Submitted By:

Sharad Chandel	**2211981360**
Rudar Attri	**2211981320**
Rudar P.S	**2211981321**
Shaiv Sud	**2211981358**

Acknowledgements

This project, "**ArtCraft: An E-Commerce Platform for Art and Craft Supplies,**" completed as part of **CS183 – Lab Oriented Project (LOP)** under the **Department of Computer Science and Engineering, Chitkara University**, would not have been possible without the support, guidance, and encouragement of numerous individuals.

I express my sincere gratitude to **Mr. Sandeep Sir**, Faculty, CSE Department, Chitkara University, for providing consistent guidance and for teaching the foundational and advanced concepts of **Frontend Development** and Cloud Computing throughout the semesters. His expertise, timely feedback, and mentorship played a significant role in shaping the technical direction of this project.

I also extend heartfelt thanks to **Ms. Rakhi Ma'am**, Faculty, CSE Department, Chitkara University, for her valuable instruction and support in **Backend Development**. Her teaching helped strengthen core understanding of server-side development, APIs, and database integration, all of which were crucial while building this platform.

I am grateful to the **Department of Computer Science and Engineering** for offering the academic environment, laboratory resources, and structured curriculum that helped in the practical implementation of full-stack technologies.

My sincere appreciation goes to my **family and friends** for their continuous encouragement, motivation, and emotional support during the development of this project. Their belief in my abilities helped me stay focused and driven throughout the process.

Finally, I acknowledge the open-source community and development tools—including **VS Code**, **Git**, **GitHub**, and the **Fedora Linux** ecosystem—whose documentation and collaborative nature significantly eased the learning and development experience.

This project stands as a culmination of all the support and knowledge shared by the above individuals and resources, and I extend my deepest gratitude to all of them.

Preface

This report presents the development of **ArtCraft**, an e-commerce platform designed to support the online sale and management of art and craft supplies. The project was completed as part of **CS183 – Lab Oriented Project (LOP)** in the **7th Semester (December 2025)** at **Chitkara University**.

Purpose and Motivation:

The primary objective of ArtCraft is to provide a **simple, user-friendly, and efficient e-commerce system** that can be used not only by professional sellers but also by **general store owners** who may lack technical expertise. Additionally, this project aims to serve as an **open-source learning resource**, enabling other students and developers to study, reuse, or extend its implementation.

1. Scope and Features:

The platform includes essential functionalities such as product management, user authentication, an admin dashboard, order handling, and a clean, accessible user interface. The goal was to integrate core concepts of full-stack application development into a cohesive, practical system.

2. Technologies Used:

The project was built using the **MERN Stack** (MongoDB, Express.js, React.js, Node.js) along with **JWT-based authentication** for secure access management. The development workflow utilized **VS Code**, **Git**, **GitHub**, and the **Fedora Linux** environment.

3. Challenges and Learning Outcomes:

Key challenges included learning the complete flow of full-stack development, understanding component-based architecture, implementing authentication, managing state across the system, and coordinating backend–frontend interactions. Overcoming these challenges strengthened understanding of modern web technologies and reinforced practical skills essential for real-world development.

This document provides a comprehensive overview of the system's design, architecture, functional components, and implementation details, reflecting a formal and technical academic presentation aligned with university standards.

Table of Contents

Table of Contents

Introduction to the Project

Introduction

This project, **ArtCraft: An E-Commerce Platform for Art and Craft Supplies**, is developed as a full-stack implementation of modern web development concepts learned throughout the semester. The inspiration behind building this platform comes from the desire to apply practical knowledge of **frontend**, **backend**, and **database** technologies, while simultaneously creating an **open-source** solution licensed under **GPL v3**.
The introduction also aligns with current **digital adoption trends** in India, where small businesses and local store owners are rapidly transitioning to online platforms. By combining motivation with technical depth, this project demonstrates how full-stack development skills can be used to create scalable and user-friendly systems accessible to a wide range of users.

Problem Statement

Many small art and craft sellers in India face significant challenges in establishing an online presence. Existing e-commerce solutions often come with **high fees**, **complex dashboards**, or **paid software packages**, making them unaffordable for small general stores. At the same time, customers frequently struggle to find affordable and reliable online platforms for art and craft supplies, resulting in dependency on larger marketplaces that charge higher commissions. There is a need for a **simple**, **affordable**, and **customizable** platform that supports essential features such as easy product management, inventory control, and order handling, while reducing the barriers for sellers entering the digital marketplace.

Objectives

Technical Objectives:

- Implement a full-stack platform using the **MERN Stack** with **JWT Authentication**.
- Provide inventory features including **add product**, **delete product**, and **order management**.
- Ensure responsive design and scalable architecture for future enhancements.

- Integrate an **AI-based floating chatbot assistant** to guide users through the platform.

User-Focused Objectives:

- Build an easy-to-use **admin dashboard** for managing products and orders.
- Simplify **seller onboarding** for store owners with minimal technical knowledge.
- Offer an **open-source learning resource** for students and developers.
- Deliver a clean, easy-to-navigate UI with toast notifications and simple design elements.

Scope of the Project

The current scope includes features on both the **admin** and **customer** sides:

Admin Side:

- Add, update, and delete products.
- Manage orders and track order status.
- Access an integrated **AI floating chatbot assistant** for support.

Customer Side:

- Browse art and craft products.
- Place orders using the **Cash on Delivery (COD)** option.
- Interact with a clean, minimal UI and receive real-time feedback via toast notifications.

Future Enhancements:

- Integration of **UPI online payment** systems.
- Additional analytics features for sellers.
- Expansion of chatbot functionalities.

Target User

- **Art and craft store owners** looking for a simple and affordable online selling platform.
- **General store owners** who want a lightweight dashboard without purchasing expensive software.
- **Students and learners** exploring full-stack development concepts.
- **Customers** purchasing art and craft supplies online.

Unique Value Proposition

ArtCraft differentiates itself from traditional e-commerce systems through:

- An integrated **AI floating chatbot** offering intelligent assistance.
- A fully **open-source** codebase available on GitHub under GPL v3, encouraging learning and customization.
- A simplified, intuitive UI/UX designed for effortless navigation with toast-based notifications.
- A strong focus on **affordability**, **simplicity**, and **customizability**, making it ideal for small businesses and independent sellers.

Feasibility Study

Technical Feasibility

The project is technically feasible due to its use of reliable, modern, and open-source technologies. The platform is built using the **MERN Stack** (**MongoDB**, **Express.js**, **React.js**, **Node**.js) with **JWT authentication** to manage secure access. Development was performed using **VS Code**, **Git**, **GitHub**, and a **Linux environment**, ensuring compatibility, flexibility, and efficient workflow management.

Deployment was done on **Render.com** using a traditional server setup. Although Render provides a free tier, the backend experiences delays of **30–60 seconds** during cold starts and goes to sleep after **15 minutes** of inactivity. Despite this, the system runs reliably once active. Alternatives such as **Vercel Serverless Functions** or future deployment on **AWS** can eliminate such delays and support greater scalability.

The entire system is lightweight and compatible with low-end hardware, making it accessible to students, learners, and small businesses. The chosen tech stack provides significant room for future upgrades, including custom domains, cloud hosting, and performance enhancements.

Operational Feasibility

The platform is designed for **store owners**, **students**, and **admin users**, all of whom can operate the system with minimal training. The admin dashboard has a clean, simple layout with straightforward controls such as **add product**, **delete product**, and **manage orders**, ensuring a low learning curve.

A short **5–10 minute training session** is sufficient for new users to understand the core functionalities. Operational ease is further enhanced through the integrated **AI floating chatbot**, implemented using the free Gemini API, which provides system guidance and answers user queries.

Because the system is browser-based, requires no installation, and uses intuitive UI components, overall operational feasibility is high.

Economic Feasibility

The project is **economically viable** due to its complete reliance on open-source and free resources. Development tools such as VS Code, GitHub, MongoDB Atlas (free tier), and Render.com were used without any monetary cost.

For sellers, this platform eliminates the need to purchase expensive e-commerce software or pay high commissions to major online marketplaces, offering significant **cost savings**. For students, it serves as a **zero-cost learning tool**, enabling hands-on experience in full-stack development.

Deployment was achieved using free hosting plans, ensuring negligible operational expenses during development. In the long term, the system remains cost-effective because maintenance and customization can be performed without licensing fees.

Schedule Feasibility

Development of the project took approximately **three months**, overlapping with active **placement drives** and academic commitments. These factors introduced delays; however, the team managed the schedule efficiently and maintained steady progress throughout the semester.

The work was completed within the academic timeline for the Lab Oriented Project (LOP) course. Tasks such as frontend design, backend development, authentication, chatbot integration, and deployment were carried out in phases, allowing the project to stay aligned with submission requirements despite external constraints.

System Analysis

Current System Overview

Most small art and craft sellers still rely on **fully offline processes**, where inventory is tracked manually and orders are handled on paper or through simple phone communication. Even where digital tools exist, sellers often use **overpriced software solutions** or outdated dashboards with poor UI/UX, which reduces efficiency and discourages adoption.

From the **seller's perspective**, the major issues are high subscription costs, outdated interfaces, and the lack of a simple, modern system tailored for small businesses. From the **customer's perspective**, limited online availability and inconsistent digital presence make it difficult to find affordable art and craft supplies through local stores.

Proposed System Overview

The proposed system introduces a **web-based, easy-to-use, and modern e-commerce platform** providing digital presence and simplified management tools for sellers. It includes an **admin dashboard**, **product management**, **order management**, an **AI chatbot assistant**, and **JWT-based authentication** to ensure secure access.

The system aims to reduce operational effort, modernize seller workflows, and offer a clean interface for both administrators and customers. It focuses on **ease of use**, **automation**, and enabling small businesses to establish an online identity with minimal technical overhead.

System Requirement

Hardware Requirements:

- A device with at least **2GB RAM** and a modern web browser.
- Compatible with both **laptops** and **mobile devices** for sellers and customers.

Software Requirements:

- A web browser (Chrome, Firefox, or equivalent).
- Internet access to use the platform.
- Deployment supported through **Render.com** for backend hosting and live API access.

Functional Requirements

- Add new products from the admin dashboard.
- Delete or update existing products.
- Manage and update order status.
- Secure login and authentication for admins.
- Customer-facing product browsing.
- Integrated **AI chatbot** for assistance.
- Toast notifications for real-time feedback.
- Admin login system for protected access.

Non-Functional Requirements

- **Usability:** Simple, intuitive interface suitable for non-technical store owners.
- **Scalability:** Ability to expand features and migrate to improved hosting in the future.
- **Performance:** Smooth browsing and fast UI response, with optimization planned beyond free-tier hosting limitations.
- **Security:** Protected routes, JWT authentication, secure data handling, and controlled admin access.
- **Future Enhancements**: Improved hosting (e.g., AWS), online payment integration, and expanded features.

Technologies Used

MongoDB

MongoDB Atlas (free tier) is used as the cloud database for the project. Its **document-oriented model**, **horizontal scalability**, and **flexible schema** make it suitable for dynamic e-commerce data.

The project uses one database (**artcraft**) containing three core collections:

- **users**
- **products**
- **orders**

  Cloud hosting ensures reliability and ease of remote access during development and deployment.

Express.js

Express.js is used as the backend framework responsible for handling **routing**, **middleware**, and **REST API** endpoints. It provides a lightweight structure for request handling, validation, authentication, and API responses.

The framework enables clean separation of concerns and supports modular API development for authentication, product handling, and order management.

React.js

React.js powers the frontend through a **component-based architecture**, enabling efficient UI rendering and code reusability.

The interface is styled using **Tailwind CSS**, providing a clean, responsive, and modern layout. State management is handled using **React Context API**, ensuring simple, predictable state flow without adding unnecessary complexity.

Node.js

Node.js serves as the runtime environment for executing backend JavaScript, supporting **asynchronous operations**, database communication, and HTTP request handling. It manages the backend server, environment configurations, and npm scripts such as:

- `npm run build`
- `npm run dev`
- `node server.js`

Environment variables include:

- `MONGODB_URI=<mongodb_uri>`
- `JWT_SECRET=<jwt_secret>`
- `GEMINI_API_KEY=<api_key>`
- `PORT=5000`
- `NODE_ENV=production`

Additional Tools

Postman: Used for testing, validating, and debugging API endpoints.

Git & GitHub: Provide version control, backup, collaboration, and open-source hosting of the complete project.

JWT (JSON Web Token): Enables secure authentication, protecting admin routes and sensitive operations.

npm: Manages all project dependencies, scripts, and development workflows.

Linux OS & VS Code: Used as the main development environment and editor for writing, debugging, and managing project code.

Render.com: Hosts and deploys the backend, offering a free-tier environment for running the server.

System Architecture

MERN Architecture

The system follows the standard **MERN** layered architecture with an added AI assistant component:

- **Client (React + Tailwind CSS)** — Single Page Application (SPA) built from reusable components (product listing, admin dashboard, forms, toast notifications). Responsible for rendering UI, capturing user actions, and calling backend APIs.
- **Server (Node.js + Express.js)** — Single backend application that exposes REST APIs, handles authentication (JWT), business logic (product/order management), and serves the built frontend (`index.html`) in a monolithic deployment.
- **Database (MongoDB Atlas)** — Cloud-hosted document store containing three collections (`users`, `products`, `orders`) for persistent storage.
- **AI Chatbot (Gemini API integration)** — External conversational assistant called from the frontend; backend mediates any required server-side calls or tokens.
- **Authentication (JWT)** — Token-based authentication issued by the server on successful login. The client stores the JWT in **localStorage** and attaches it to authenticated API requests.

Client–Server Interaction Flow

1. User / Admin Login:

1. User submits credentials via the login form (frontend).
2. Frontend sends `POST /api/auth/login` with credentials.
3. Server validates credentials, generates a **JWT**, and returns it in the response.
4. Frontend stores the JWT in **localStorage** and sets an authorization header (`Authorization: Bearer <token>`) for subsequent requests.
5. Protected API endpoints verify JWT in incoming requests; if valid, the server authorizes the action.

2. Admin — Product Management (Add / Delete / Update):

1. Admin uses the dashboard UI to add/update/delete a product.
2. Frontend sends the corresponding authenticated request (`POST/PUT/DELETE /api/products`) with the JWT header.
3. Server validates JWT, runs input validation and business logic, then updates MongoDB (`products` collection).
4. Server responds with operation status; frontend displays toast notifications and updates UI state via React Context API.

3. Customer — Browse & Place Order

1. Customer browses products; frontend fetches product data via `GET /api/products`.
2. Customer places an order through the UI (checkout with COD). Frontend sends `POST /api/orders` with order details and optional user token.
3. Server validates the order, stores it in the `orders` collection, returns confirmation.
4. Admin views and updates order status through the dashboard; frontend polls or fetches order updates as needed.

4. AI Chatbot Interaction:

1. User clicks the floating chatbot; frontend sends user query to the chatbot endpoint.
2. Depending on implementation, the request may go directly to the external Gemini API (client-side) or via the backend which injects/guards API keys.
3. Chatbot returns contextual responses; frontend renders replies in the floating UI. Chatbot also assists with system usage (how to add products, check orders).

API Architecture

APIs are grouped modularly by function. Each module follows REST principles and enforces authentication where required.

- **Auth Module**
    - `POST /api/auth/register` — create user/admin (if applicable)
    - `POST /api/auth/login` — validate credentials, return JWT

    - JWT issued with short-to-medium lifetime; server-side middleware validates token on protected routes.

- **Products Module**
    - `GET /api/products` — public product listing
    - `POST /api/products` — add product (admin, authenticated)
    - `PUT /api/products/:id` — update product (admin)
    - `DELETE /api/products/:id` — delete product (admin)
- **Orders Module**
    - `POST /api/orders` — create new order (customers)
    - `GET /api/orders` — admin view of orders (authenticated)
    - `PUT /api/orders/:id` — update order status (admin)
- **Chatbot / Support Module**
    - `POST /api/chat` — relay user messages to AI assistant (backend may mediate API key usage)
    - Designed to be stateless or minimally stateful depending on required context persistence.

Security notes: Protected endpoints require the `Authorization` header with a valid JWT. Sensitive keys and URIs are provided via environment variables on the server; the client never stores raw secret values.

Deployment Architecture

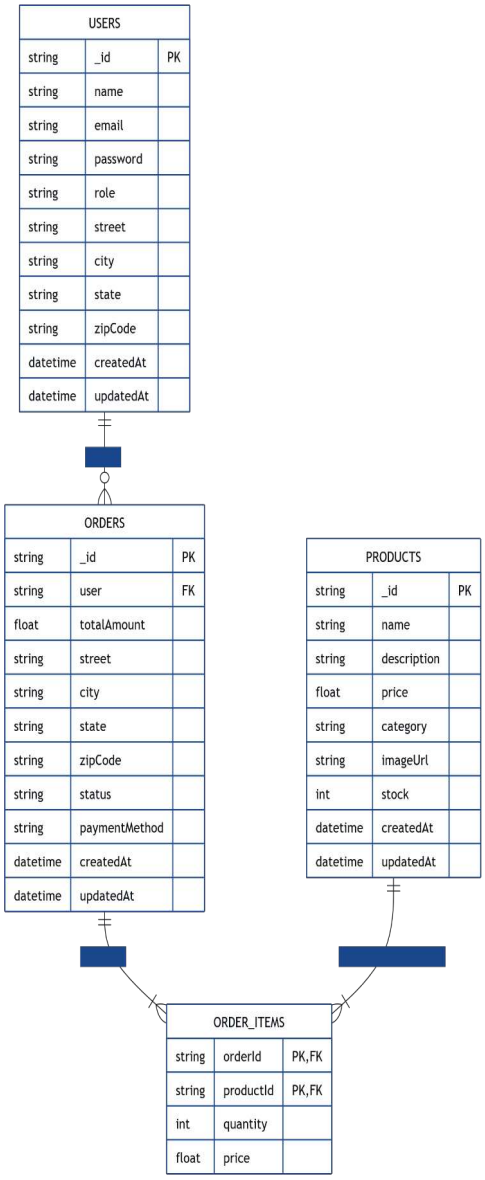
The system is deployed as a **monolithic application**, where the React frontend is built using `npm run build` and served directly by the Express backend. The backend and static frontend files run together on **Render.com**, using environment variables for all sensitive configurations.

Data is stored on **MongoDB Atlas**, connected through secure environment variables. Source code and deployment updates are managed through **GitHub**, allowing quick redeploys.

Although optimized for simplicity, the architecture can scale further by migrating to **AWS** or separating the frontend and backend into independent services for improved performance and availability.

Database Design

ER Diagram



Collection Schema Design

1. Users Collection:

Fields include:

- **name**: String
- **email**: unique identifier
- **password**: hashed password (securely stored in real implementation)
- **role**: user/admin
- **address**: optional embedded object
- **timestamps**: createdAt, updatedAt

2. Products Collection:

Fields include:

- **name, description, price, category**
- **imageUrl**: static asset or link
- **stock**: numerical quantity available
- **timestamps**

3. Orders Collection:

Fields include:

- **user**: ObjectId referencing Users
- **items**: Array of product, quantity, price
- **totalAmount**
- **shippingAddress** (embedded)
- **status** (pending → delivered)
- **paymentMethod**
- **timestamps**

Relationships between Collections

- **Users → Orders**: Each order links to one user via `user: ObjectId`.

- **Orders → Products**: Each order contains an array of `items`, where each item references a `product: ObjectId`.
- **Products** are independent documents but frequently appear inside Orders as references.
- No direct relationship is stored between Products and Users; Orders serve as the join link between them.

This structure ensures:

- Flexible growth of product catalog
- Efficient querying for admin dashboards
- Clean separation of user, product, and transactional data

Sample Documents (JSON)

Sample order document:


```
{
 "_id": "682a311859646c445b965fcb",
 "user": "682a301da0f4b93bcf68df28",
 "items": [
  {
   "product": "682a301da0f4b93bcf68df2b",
   "quantity": 1,
   "price": 9.99
  },
  {
   "product": "682a301da0f4b93bcf68df2a",
   "quantity": 1,
   "price": 24.99
  }
 ],
 "totalAmount": 34.98,
 "shippingAddress": {
  "street": "L.I.G BLOCK-6, A-3",
  "city": "Parwanoo",
  "state": "HP",
```

```
    "zipCode": "173220"
  },
  "status": "delivered",
  "paymentMethod": "Cash on Delivery",
  "createdAt": "2025-05-18T10:12:24.375Z",
  "updatedAt": "2025-08-14T12:45:27.333Z"
}
```


Sample product document:


```
{ "_id": "6902dabcf91a87c9c5fe1985",
  "name": "Acrylic Paint Set",
  "description": "Professional 12-color acrylic paint set, perfect for canvas painting",
  "price": 99.99,
  "category": "Painting",
  "imageUrl": "/uploads/acrylic-paint.jpg",
  "stock": 100,
  "createdAt": "2025-10-29T08:45:48.733Z",
  "updatedAt": "2025-10-29T17:47:21.111Z" }
```


Sample user document:


```
{
  "_id": "6902dabcf91a87c9c5fe1983",
  "name": "Sharad",
  "email": "sharad@gmail.com",
  "password": "123",
  "role": "user",
  "createdAt": "2025-10-29T08:45:48.622Z",
  "updatedAt": "2025-10-29T08:45:48.622Z"
}
```

Modules of the Project

User Module

The User Module manages all functionalities related to customer authentication and basic account operations. Users can **register** and **log in** to access protected features such as the cart, checkout, and order history. After login, users can **view their profile**, but profile updates are not yet supported; during checkout, users manually enter address information each time.

Role-based access control allows standard users to access shopping features, while an **admin role** gains additional privileges such as dashboard visibility and management options. Email verification is not required in the current version but is planned for future implementation.

Product Module

This module handles all product-related operations in the system. Customers can browse available items and open a **product details page** for additional information. The platform currently does not support search or filtering features, and all products are treated as **simple single-item entries** without variations.

Admins can perform full product management through the dashboard, including **adding new products**, **editing existing products**, and **deleting products** as needed.

Cart & Checkout Module

The Cart Module enables users to manage selected items before placing an order. Customers can **add products to the cart**, **remove items**, and **update quantities** directly from the cart interface. During checkout, users provide their **street address, city, state, and zip code**, along with selecting a payment method. Currently, the system supports **Cash on Delivery (COD)**, with **UPI-based online payments planned** for future versions. An order summary is shown before final confirmation, ensuring clarity and transparency.

Orders Module

This module manages all order lifecycle operations. Customers can view a list of their **past orders**, check **order details**, and track **order status** as it progresses from pending to delivered. Admins have advanced capabilities, including viewing **all orders**, changing order status, and managing cancellations. A cancellation option exists, but it is currently restricted to the admin. Refund handling is also managed manually by the admin or store owner, depending on the situation.

Admin Module

The Admin Module provides system-wide management tools accessible only to authenticated users with the **admin role**. Admins can access a dedicated **dashboard**, which includes options for managing products, viewing all orders, updating order statuses, and controlling cancellations. The module focuses on simplicity and functionality, offering clean, straightforward interfaces without additional analytics or visual reporting at this stage. All admin actions are protected through role-based authentication to ensure system security and data integrity.

Payment Integration Module

The current system supports only **Cash on Delivery**, which simplifies deployment and avoids the complexities of integrating payment gateways. Future enhancements include full **UPI integration**, with plans to implement **Razorpay** as the primary payment provider.
The main limitation is that modern payment platforms now require extensive business verification and documentation before enabling UPI or online payments, making integration more difficult for student or early-stage academic projects. Once verification requirements can be met, the platform will expand to support secure online payments.

System Design

This section presents the key system design diagrams created for the ArtCraft E-Commerce Platform. These visuals illustrate how the system components interact, how data flows between modules, and how users engage with the platform. The diagrams included are:

1. Context Diagram (Level 0)
2. Data Flow Diagram (DFD)
3. Flowchart
4. Sequence Diagram

Context Diagram (Level 0)

This diagram provides a high-level overview of the entire system, showing all external entities interacting with the ArtCraft System.

It highlights the primary inputs and outputs between:

- Customer User
- Admin
- Payment Gateway (future)

- MongoDB Database

It establishes the system boundary and clarifies the responsibilities handled by the platform.

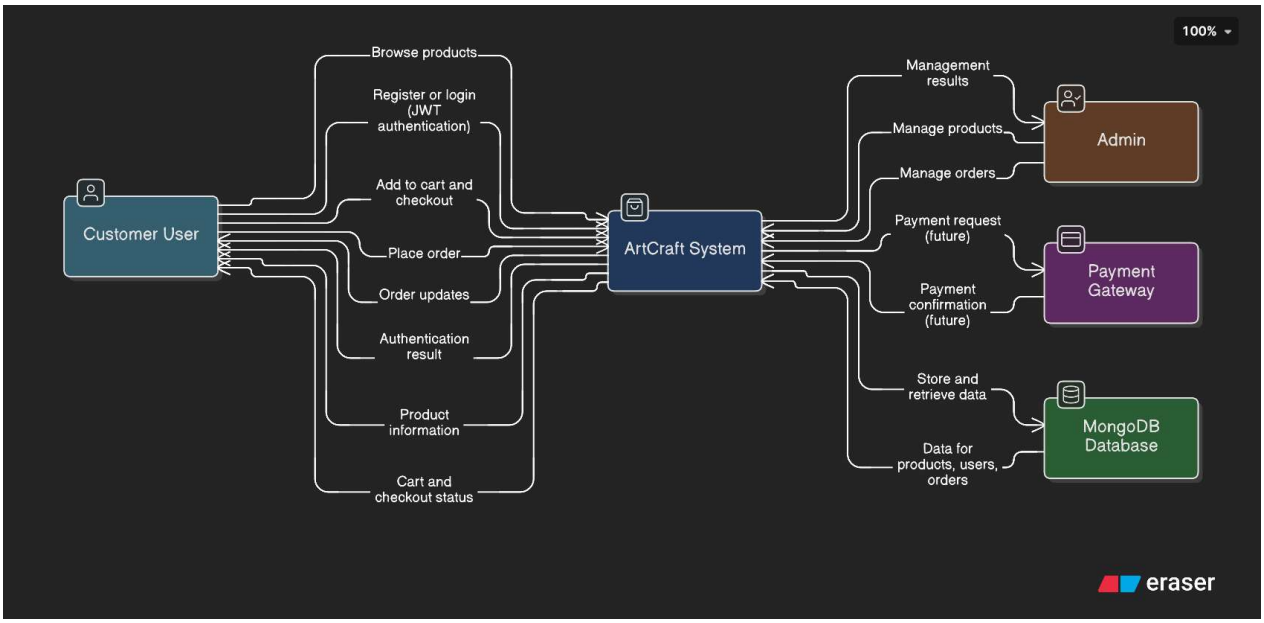


Figure: Context Diagram

Data Flow Diagram (DFD)

This diagram breaks the system into logical subprocesses and shows how data moves across:

- User Management
- Product Management
- Order Processing
- Cart & Checkout
- Admin Operations
- Database Operations
- Future Payment Flow

It outlines the transformation of data between components and helps explain system workflow at a structured level.

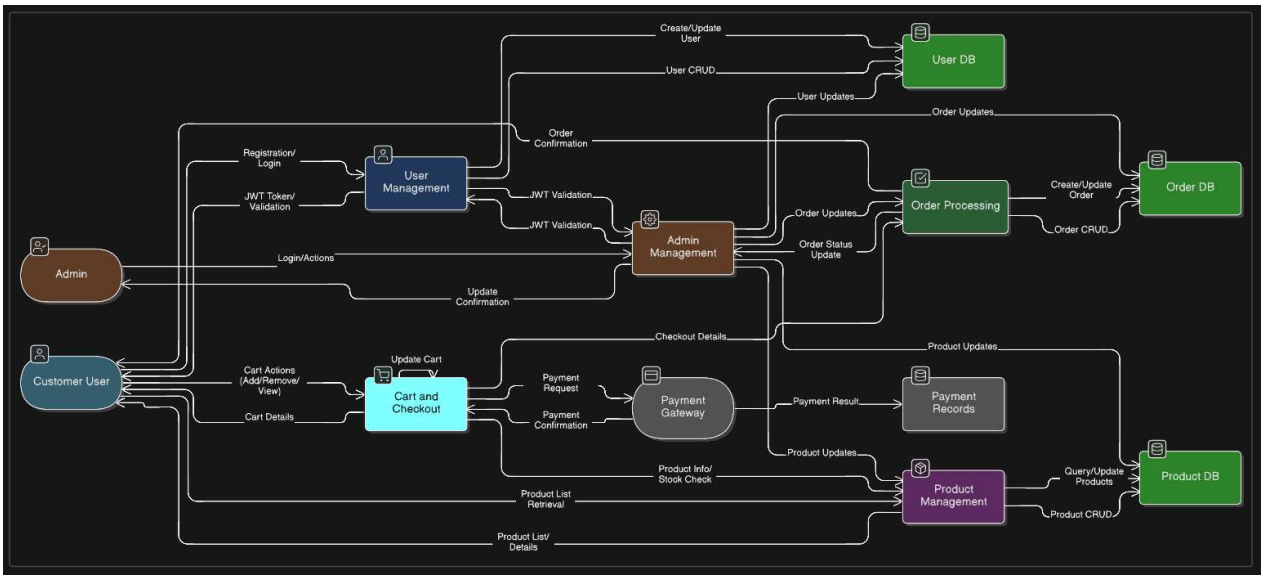


Figure: Data Flow Diagram

Flowchart

The flowchart represents two major journeys:

Customer Interaction Flow:

- Register/Login
- Browse products
- View details
- Add to cart
- Checkout
- Payment selection
- Order completion

Admin Interaction Flow:

- Admin login
- Add/Edit/Delete products

- View orders
- Update or cancel orders

It visually maps step-by-step decision flows for both users.

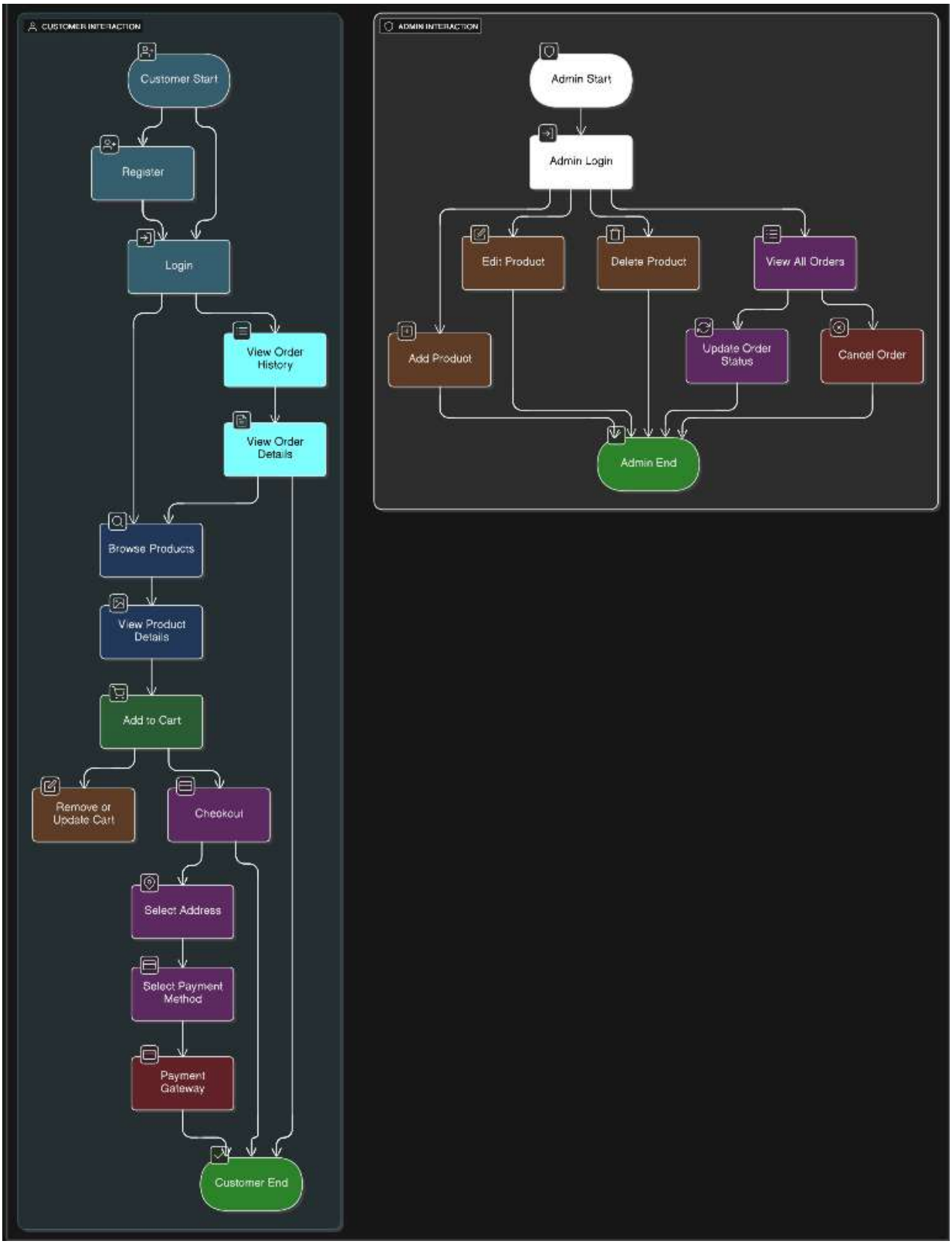


Figure: Flowchart

Sequence Diagram

This diagram captures the chronological interaction between:

- Customer
- Web App
- Payment Service
- Inventory System

Operations illustrated include:

- Browsing products
- Adding items to cart
- Checkout and conditional payment flow
- Inventory update sequence
- Order and stock confirmation

The diagram helps clarify real-time request/response behavior in the system.

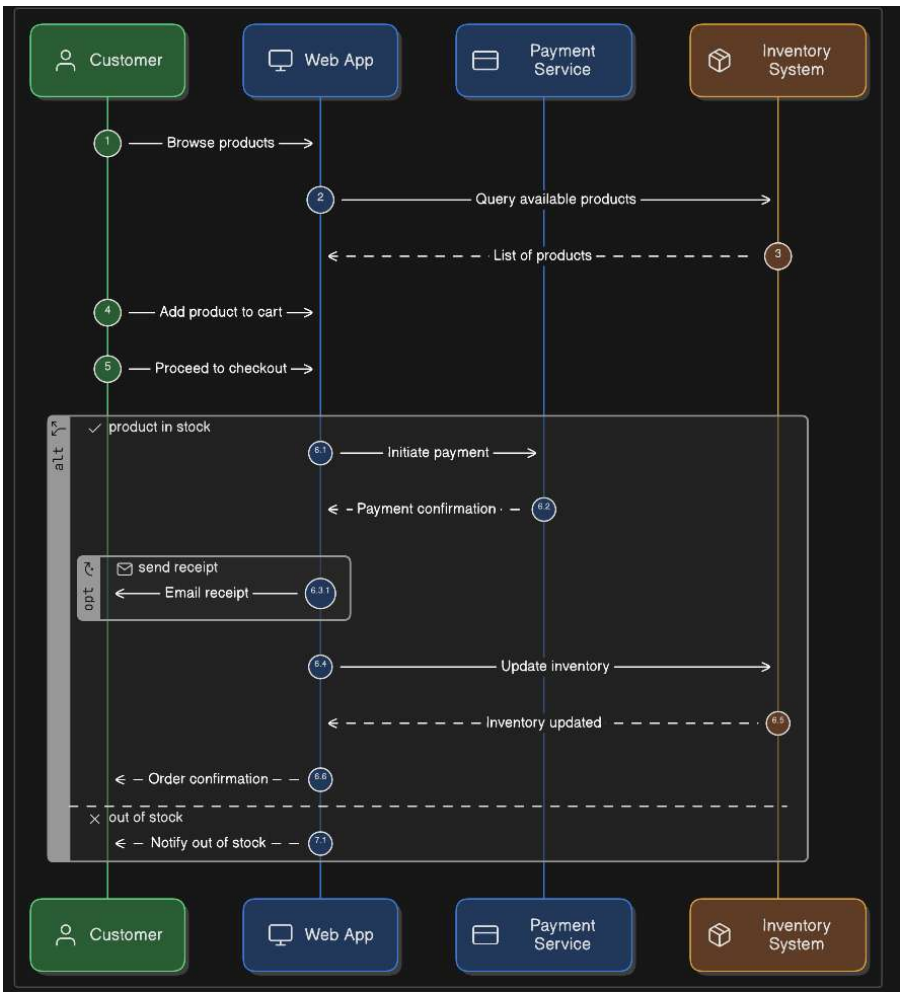


Figure: Sequence Diagram

UI/UX Design

Wireframes

Although initial wireframes were created, they are intentionally excluded from this documentation. The final UI closely follows standard e-commerce patterns: a clean homepage hero section, product grid listing, product details layout with image + information split, simplified cart and checkout forms, and a clearly structured admin dashboard for product and order management.

Color Palette and Typography

Color Palette:

The interface primarily follows TailwindCSS's built-in palette, resulting in a modern, clean, and accessible UI. Key colors extracted from the implemented components include:

- **Primary Colors**
    - **Blue-500 (#3B82F6)** – Buttons, CTAs, accents
    - **Indigo-600 (#4F46E5)** – Gradient hero background
- **Secondary / Utility Colors**
    - **Gray-50 / Gray-100 / Gray-600 / Gray-800** – Backgrounds, borders, text
    - **Green-100 / Green-800** – Stock availability badge
    - **Red-100 / Red-800** – Out-of-stock badge
- **UI Feedback Colors**
    - **Blue-700** – Hover effect for primary buttons
    - **White (#FFFFFF)** – Hero text, button foregrounds
    - **Custom Fallback Image Handling Color Logic** – Used via utility classes

These colors help maintain a balanced visual hierarchy and ensure readability across screens.

Typography:

The application uses a system-optimized, performance-friendly font stack:

-apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, 'Helvetica Neue', Arial, sans-serif

Characteristics:

- Highly legible on both desktop and mobile.
- Automatically uses platform-native UI typography (e.g., San Francisco on macOS/iOS, Segoe UI on Windows).
- No external font loading improves performance and Core Web Vitals.

Tailwind's consistent spacing utilities (p-4, mb-6, gap-4, rounded-lg) and shadows (shadow-md, shadow-sm) create a uniform, clean aesthetic across components.

Responsiveness Strategy

Mobile-First Structure:

The UI follows Tailwind's mobile-first philosophy:

- Base styles target small screens.
- Layout expands using responsive classes (sm:, md:, lg:).

Examples:

- Product grid transitions from **1 → 2 → 3 → 4 columns** across breakpoints.
- Flex layouts convert automatically with flex-col sm:flex-row (used in the hero buttons).
- Buttons and spacing adjust proportionally without custom CSS.

Practical Responsive Adjustments:

Key responsive behaviors include:

1. **Navbar and Navigation**
    - Clean top navigation layout adapts for mobile stacking.
    - Login/Register buttons remain accessible even on small screens.
2. **Product Cards**
    - Height-limited product images (max-h-44) ensure visual consistency.
    - Text truncation (line-clamp) maintains clean layout on narrow screens.

3. **Checkout Workflow**
    - Form components stack vertically on mobile.
    - Spacing and padding adapt automatically via Tailwind utility classes.
4. **Admin Dashboard**
    - Tables and management panels scroll horizontally on small devices.
    - Cards for metrics remain responsive using flex/grid utilities.

Device Testing:

Testing was performed on:

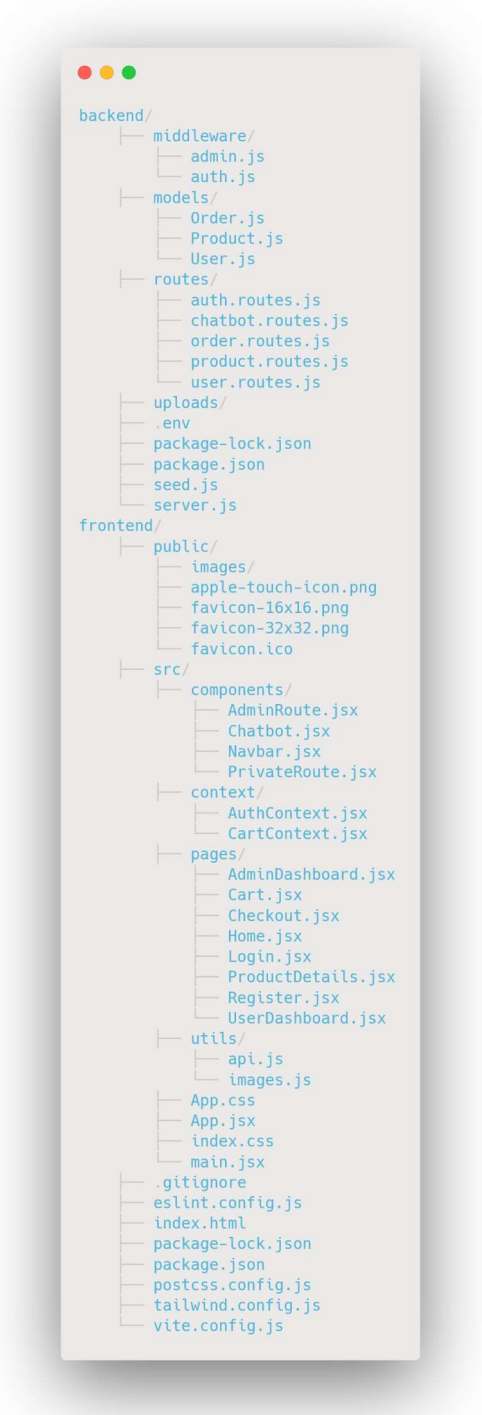
- **Desktop (primary)**
- **Mobile devices**

  Adjustments were made using Tailwind's responsive utilities to fix spacing, overflow, and layout issues. Tablet optimization is not explicitly implemented but inherits responsive behavior from existing breakpoints.

Implementation/Coding

This section provides a structured and technical overview of the project implementation. It highlights the core folder structure, essential backend and frontend files, and the most relevant code snippets. All samples are intentionally short and focused on key logic.

Folder Structure



Key Backend Files (API, Models, Controllers)

Authentication (auth.routes.js): Handles register, login, and fetching authenticated user.

```
// Register
router.post('/register', async (req, res) => {
  const { name, email, password } = req.body;
  const exists = await User.findOne({ email });
  if (exists) return res.status(400).json({ message: 'User exists' });

  const user = await User.create({ name, email, password });
  const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET);
  res.json({ token, user });
});
```

Figure: Register Endpoint Snippet

```
exports.auth = (req, res, next) => {
  const token = req.headers.authorization?.split(' ')[1];
  const decoded = jwt.verify(token, process.env.JWT_SECRET);
  req.user = decoded;
  next();
};

exports.isAdmin = (req, res, next) => {
  if (req.user.role !== 'admin') return res.status(403).json({ message: 'Unauthorized' });
  next();
};
```

Figure: Middleware Snippet (auth.js)

Product API (product.routes.js): Handles product creation, listing, update, delete.

```
const upload = multer({
  storage: multer.diskStorage({
    destination: (req, file, cb) => cb(null, uploadsDir),
    filename: (req, file, cb) => cb(null, Date.now() + path.extname(file.originalname))
  }),
  fileFilter: (req, file, cb) => {
    if (!file.originalname.match(/\.(jpg|jpeg|png)$/)) return cb(new Error('Images only'));
    cb(null, true);
  }
});
```

Figure: Multer Snippet

```
router.post('/', [auth, isAdmin, upload.single('image')], async (req, res) => {
  const product = await Product.create({
    name: req.body.name,
    imageUrl: '/uploads/' + req.file.filename
  });
  res.status(201).json(product);
});
```

Figure: Create Product Snippet

Orders API (order.routes.js): Handles order creation and admin order access.

```
router.post('/', auth, async (req, res) => {
  const { items, shippingAddress, totalAmount } = req.body;

  const order = await Order.create({
    user: req.user.id,
    items,
    shippingAddress,
    totalAmount
  });

  res.status(201).json(order);
});
```

Figure: Order Creation Snippet

server.js:

```
app.use(cors());
app.use(express.json());
app.use('/uploads', express.static(path.join(__dirname, 'uploads')));

mongoose.connect(process.env.MONGODB_URI);

app.use('/api/products', productRoutes);
```

Figure: Core Setup Snippet

Key Frontend Files (Components, Pages, Context)

Context API:

```
const login = async (email, password) => {
  const res = await api.post('/api/auth/login', { email, password });
  localStorage.setItem('token', res.data.token);
  setUser(res.data.user);
};
```

Figure: AuthContext.jsx – Login Snippet

```
const addToCart = (product, quantity = 1) => {
  setCart(prev => [...prev, { product, quantity }]);
};
```

Figure: CartContext.jsx – Add to Cart Snippet

Utility Layer:

```
api.interceptors.request.use((config) => {
  const token = localStorage.getItem('token');
  if (token) config.headers['Authorization'] = `Bearer ${token}`;
  return config;
});
```

Figure: api.js – Token Interceptor

```
export const getImageUrl = (path) =>
  path?.startsWith('http') ? path : `${UPLOADS_URL}${path}`;
```

Figure: Image URL Helper

Core Components:

```
const AdminRoute = ({ children }) => {
  const { user } = useAuth();
  return user?.role === 'admin' ? children : <Navigate to="/" />;
};
```

Figure: AdminRoute.jsx

```
const PrivateRoute = ({ children }) => {
  const { user } = useAuth();
  return user ? children : <Navigate to="/login" />;
};
```

Figure: PrivateRoute.jsx

```
{user ? (
  <button onClick={logout}>Logout</button>
) : (
  <Link to="/login">Login</Link>
)}
```

Figure: Navbar Logout Snippet

Figure: Chatbot.jsx (API Call)

Screenshots/Application Walkthrough

This section provides a visual overview of the ArtCraft E-commerce Platform, covering both desktop and mobile interfaces. Each subsection briefly explains the purpose of the page followed by the corresponding screenshots. All screenshots should be centered, labeled, and sized uniformly for clarity.

Home Page

Description:

The Home Page displays featured products, simple navigation, and a clean UI designed for quick browsing. It serves as the entry point for both new and returning users.

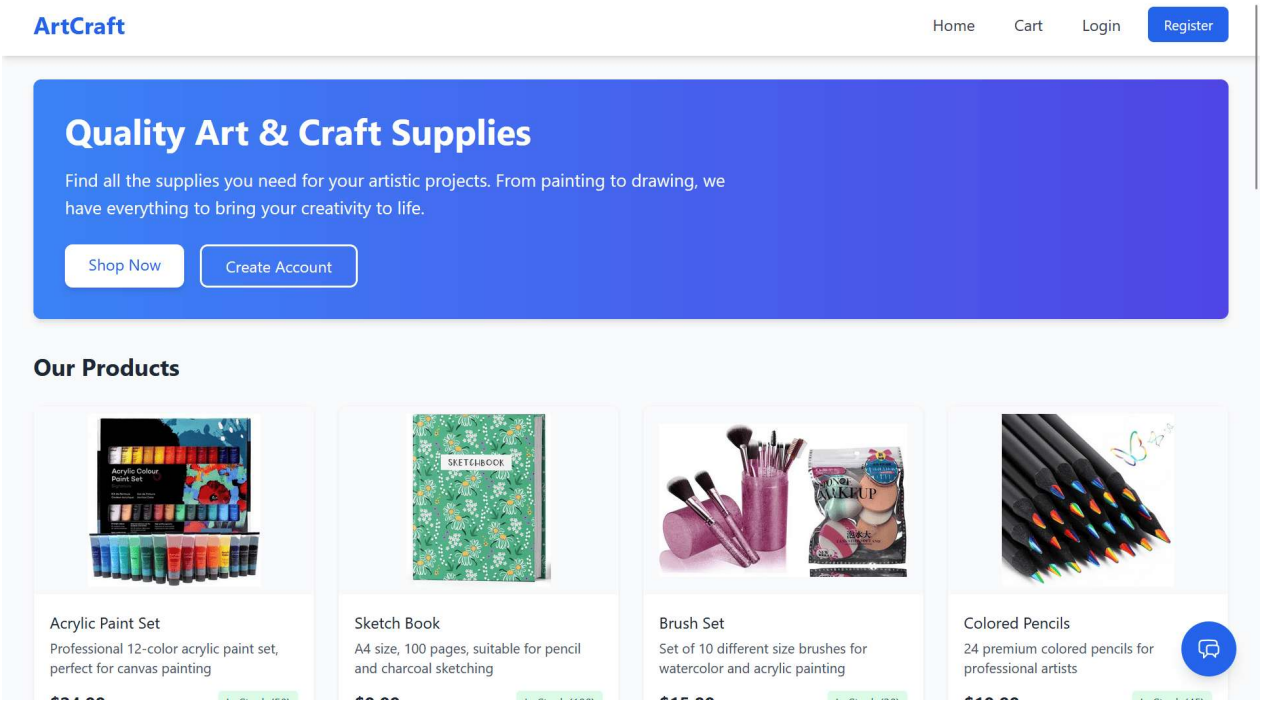


Figure 1.1: Desktop View – Home Page

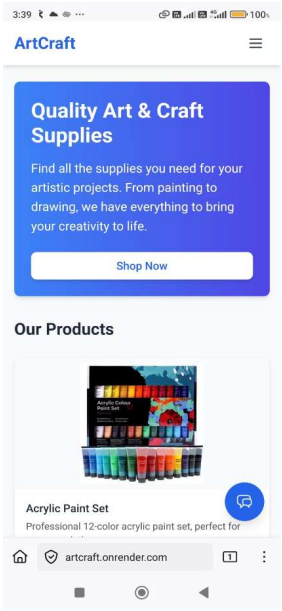


Figure 1.2: Mobile View – Home Page

Product Detail Page

Description:

Shows detailed information about a selected product, including price, description, category, stock status, and an option to add the item to the cart.

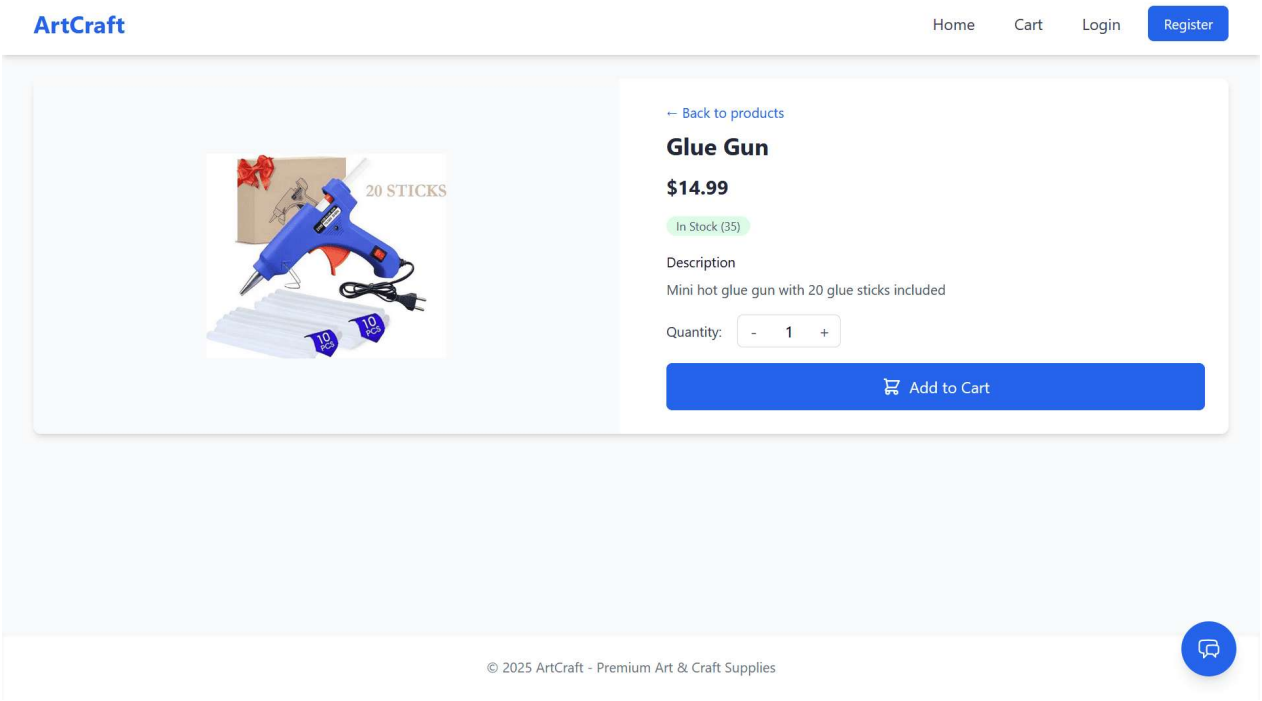


Figure 2.1: Desktop View – Product Details

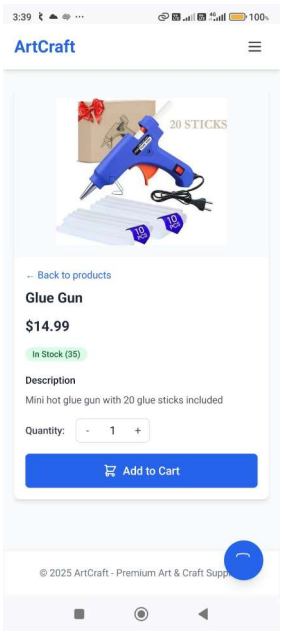


Figure 2.2: Mobile View – Product Details

Cart Page

Description:

Displays all items added by the user, allowing quantity updates, item removal, and total amount calculation.

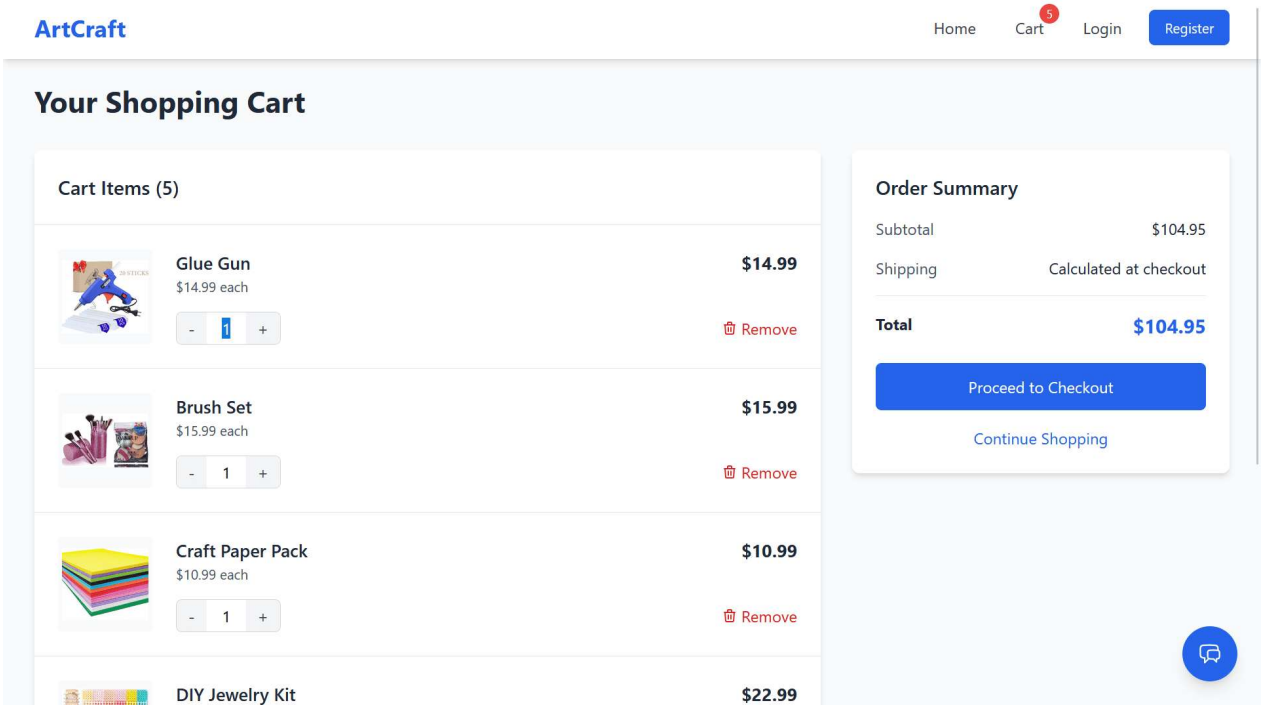


Figure 3.1: Desktop View – Cart Page

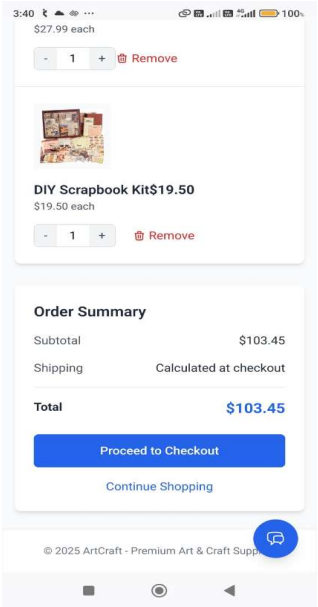


Figure 3.2: Mobile View – Cart Page

Checkout Page

Description:
Supports address input (street, city, state, zip code) and payment method selection (COD currently, UPI planned). The final summary is shown before order placement.

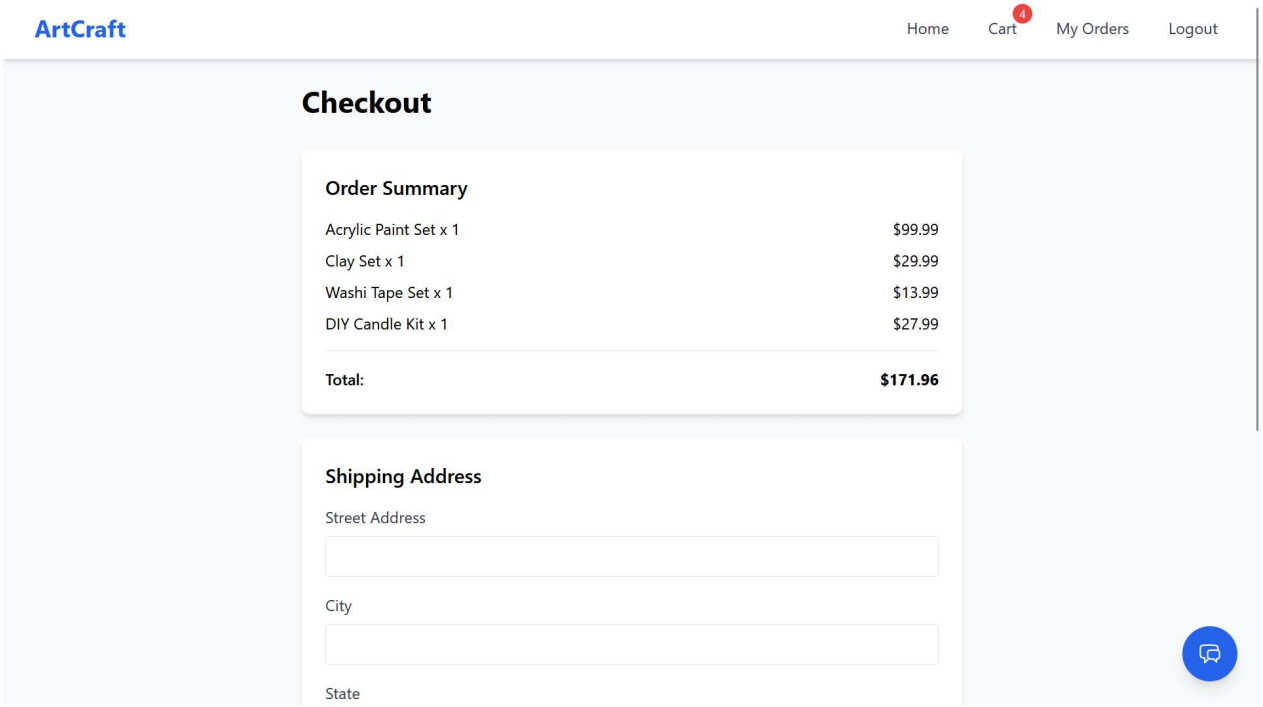


Figure 4.1: Desktop View – Checkout Page

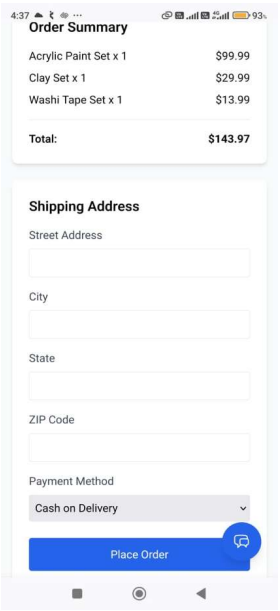


Figure 4.2: Mobile View – Checkout Page

Admin Dashboard

Description:

Admin-only interface featuring product management, order management, and status updates. Designed to be simple, functional, and easy to navigate.

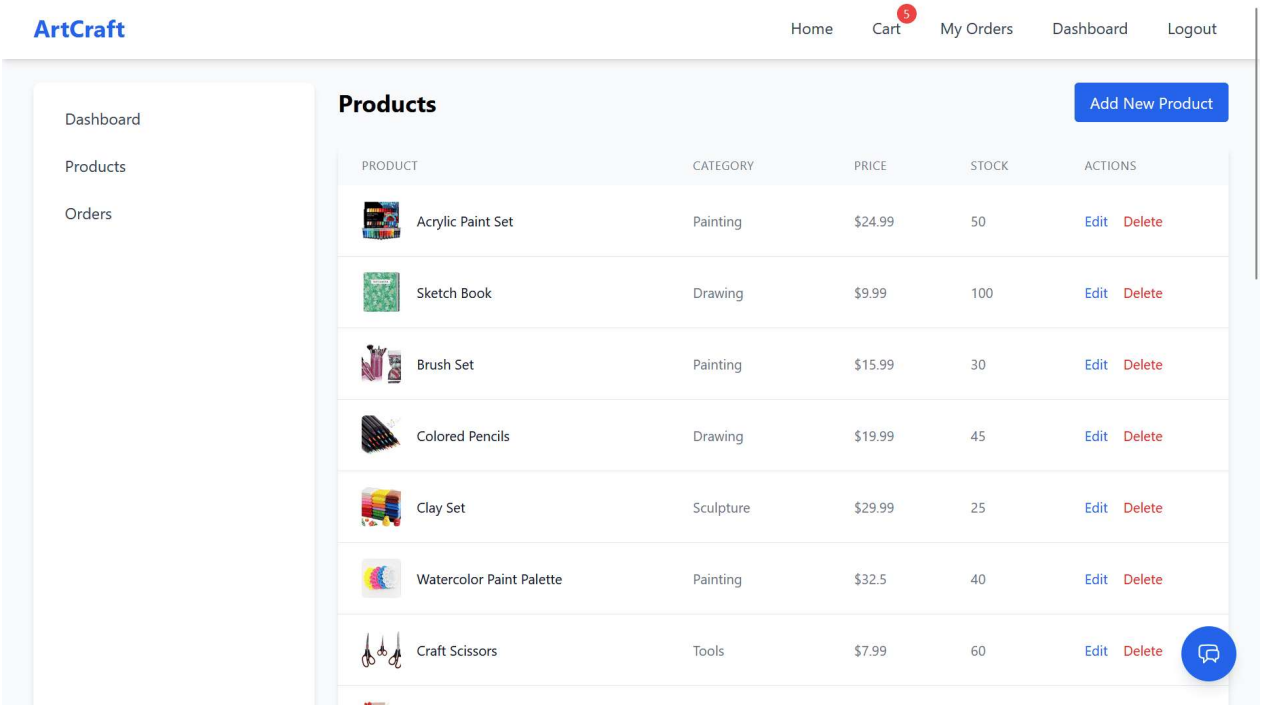
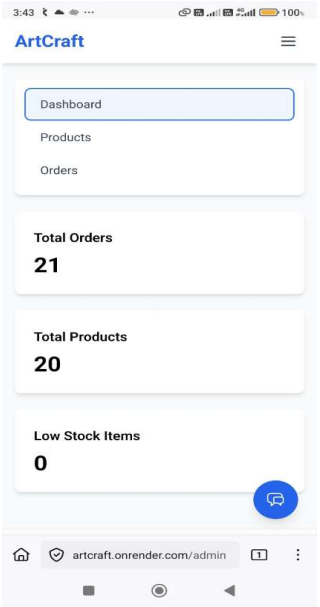


Figure 5.1: Desktop View – Admin Dashboard



- *Figure 5.2: Mobile View – Admin Dashboard*

Security Measures

Authentication & Authorization (JWT)

The application uses **JSON Web Tokens (JWT)** to authenticate users and authorize protected actions. Upon successful login, the server issues a token that is stored in the **browser's localStorage**. The token has a **1-day expiry**, after which users must log in again. There is **no refresh-token mechanism** in the current version.

Protected routes verify the JWT on every request, ensuring that only authenticated users can access restricted functionality such as checkout, viewing orders, or accessing the admin dashboard.

Password Hashing

Passwords are currently stored as **plain text** due to the academic nature of the project. While this simplifies development, it is not secure for production environments.

Industry-standard practice recommends hashing all passwords using libraries such as **bcrypt**, ensuring:

- one-way encryption
- salted hashes to prevent rainbow-table attacks
- improved security during data breaches

Future versions of the project will adopt bcrypt-based hashing.

Input Validation

The system performs input validation on **both frontend and backend** to ensure data integrity and prevent malformed requests.

Validation includes:

- empty field checks
- numeric validation (quantity, price)
- email format checks
- Mongoose schema-level validation for required fields and datatypes
  Although lightweight, this approach prevents common user input errors and enforces consistency across all modules.

API Protection

The backend implements middleware-based route guarding to secure API endpoints.
Key protections include:

- **JWT verification middleware** for all authenticated routes
- **Admin-only middleware** for restricted operations such as product management and order status updates
  These security layers ensure that unauthorized users cannot modify data or access sensitive resources.
  Rate limiting and advanced CORS security are not implemented yet but are strongly recommended for future upgrades to improve resilience against abuse and automated attacks.

Role-Based Access

Role-Based Access Control (RBAC) ensures users receive permissions appropriate to their role.

- **Admin role:** can add, edit, and delete products; manage all orders; update order statuses; and control cancellations.

- **User role:** can browse products, manage their cart, and place orders.
- **Guest users:** can browse the catalog and add items to the cart but cannot complete checkout without logging in.

  This layered structure provides clear separation of privileges and reduces the risk of unauthorized activities within the system.

Deployment

Deployment Environment

The application is deployed using a **monolithic architecture** on **Render.com**, where both the backend (Node.js + Express) and the production React build are hosted together. The React application is compiled using `npm run build` and served as static files directly from the backend.

Render manages server hosting, environment variables, and deployment logs. Future deployment plans include migrating to **AWS EC2** with a **custom domain**, which will provide greater scalability, improved performance, and better control over server resources.

Build & Department Steps

Deployment follows a simple manual workflow supported by Git and Render:

1. **Local Build & Development**
   - Frontend development: `npm run dev` (Vite)
   - Backend development: `node server.js`
   - Production frontend build: `npm run build`

2. **Repository Preparation**
   - Code is committed and pushed to GitHub for version control and backup.

- The repository contains both frontend and backend folders within a single project structure.

3. **Deploy on Render**
    - Create a new Web Service on Render.
    - Connect the GitHub repository manually.
    - Configure build command (e.g., install dependencies and build frontend).
    - Configure start command for backend: `node server.js`.
    - Upload environment variables through Render's dashboard.

4. **Monolithic Serving**
    - The backend serves the built frontend `index.html` from the `dist` or `build` folder after `npm run build`.
    - All client-side routes are forwarded to the frontend build.
    - API routes remain accessible under `/api/`.

This approach is simple, cost-effective, and avoids managing two separate hosting services.

Environment Variables

The following environment variables are used for secure configuration:

- **MONGODB_URI** = `your-mongodb-atlas-uri`
- **JWT_SECRET** = `abcd`
- **PORT** = `5000`
- **NODE_ENV** = `production`
- **GEMINI_API_KEY** = `your-gemini-api-key`

All sensitive variables are stored in Render's environment panel and excluded from public code repositories.

CI/CD

The current deployment does not use automated CI/CD pipelines. All deployments are triggered manually on Render whenever updated code is pushed to GitHub.

Future enhancements include integrating **GitHub Actions** for automated testing and deployment, enabling:

- automated builds
- test execution before deployment
- auto-deploy on commits to the main branch
  This will improve workflow efficiency and reduce deployment errors.

Challenges and Solutions

Understanding MERN Workflow

Challenge:

Learning the complete MERN workflow was initially difficult, especially connecting React (frontend) with Node/Express (backend) and managing data flow across components and APIs.

Solution:

The team gradually broke the system into smaller modules, implemented API endpoints one by one, and tested them using Postman. This modular approach clarified the flow between frontend, backend, and database.

State Management in React

Challenge:

Managing component state, shared data, and UI updates was confusing in the beginning. React's component-driven architecture required a deeper understanding of props, state, and re-rendering behavior.

Solution:

The use of **React Context API** simplified global state handling for cart data, authentication, and user session tracking. Over time, consistent patterns were developed to maintain cleaner and more predictable state logic.

API Errors & Debugging

Challenge:

Frequent API errors, mismatched data types, incorrect endpoints, and backend response issues significantly slowed progress. Some bugs were difficult to trace and almost caused the team to give up at certain stages.

Solution:

Systematic debugging using console logs, Postman testing, and step-by-step validation of API requests helped identify and resolve issues. Improved error handling in both backend and frontend reduced unexpected failures.

Mobile Responsiveness

Challenge:

Several pages broke or appeared misaligned on smaller screens during testing. UI elements were either too large, overlapping, or not adjusting properly.

Solution:

TailwindCSS responsive utilities (`sm`, `md`, `lg`) were used to redesign layouts. The team revisited all components and ensured mobile-first adjustments for better responsiveness across devices.

Floating AI Chatbot Integration

Challenge:

Implementing the floating AI chatbot using a free API was new and involved unfamiliar UI positioning, API calls, and error management.

Solution:

A minimal version was built first, then gradually enhanced. Fixed-position CSS styling was applied, and the API calls were tested separately before integrating with the UI. This step-by-step approach made integration manageable.

Backend Debugging & Data Handling

Challenge:

Most time was spent debugging backend logic, particularly order creation, product updates, and

error handling with MongoDB schemas.

Solution:

Clearer schema definitions, better console-based debugging, and consistent testing using sample data helped stabilize backend operations.

Time & Motivation Challenges During Placement Season

Challenge:

Placement season caused major delays, reduced motivation, and forced the team to slow down development. Some planned features were postponed or dropped due to time constraints.

Solution:

The team prioritized essential features, created a realistic task list, and focused on completing core modules first. Steady progress over smaller daily targets helped maintain momentum despite busy schedules.

Learning Curve with React

Challenge:

React was the hardest technology to learn, especially understanding components, routing, lifecycle concepts, and reactive rendering.

Solution:

Hands-on practice, building small sample components, and reviewing documentation helped build confidence. Eventually, the team mastered the basics needed for building the full frontend.

Future Enchancements

UPI & Online Payment Integration

A major planned enhancement is adding full **UPI-based online payments** using providers such as Razorpay. This will allow faster, secure transactions and reduce dependence on Cash on Delivery. Business verification requirements currently limit integration, but once fulfilled, the system will support seamless online checkout.

Advanced Search and Filtering

Future versions will include a more sophisticated product search system with filters for categories, price range, and popularity. This will significantly improve the browsing experience, especially for users looking for specific art and craft supplies.

Improved User Profile & Address Management

The platform will extend the User Module to allow users to **update profiles**, save **multiple addresses**, and store past delivery details. This will speed up the checkout process and provide a more personalized shopping experience.

Enhanced Admin Analytics

The admin dashboard will be expanded with analytics such as sales charts, product performance, low-stock alerts, and customer statistics. This will help store owners make data-driven decisions and improve inventory planning.

Dedicated Mobile Application

A future goal is to build a dedicated Android/iOS mobile app using React Native or Flutter. This will improve accessibility for sellers and students who prefer using mobile devices for browsing and order management.

Separate Frontend and Backend Deployment

To improve performance and scalability, the monolithic architecture can be separated into:

- **Frontend** (React) hosted on Vercel or Netlify
- **Backend** (Node/Express) hosted on AWS EC2

  This architecture will reduce cold-start delays and support larger traffic loads.

Email Verification & Password Reset

Security will be enhanced by implementing:

- email OTP verification
- password reset via email links
- stronger password policies

  These improvements will increase authentication reliability and protect user accounts.

Full Inventory Management Enhancements

Advanced inventory tracking features will be added, such as:

- automated stock alerts
- bulk product imports
- low-stock notifications for admins

  This will help store owners maintain accurate stock levels efficiently.

AI Chatbot Enhancement

The existing AI chatbot can be improved to support:

- product recommendations
- order tracking
- customer FAQs
- multilingual support

  Integrating a more robust AI backend will make the chatbot more helpful and context-aware.

Migration to AWS Infrastructure

Future versions may migrate to **AWS EC2, S3, Route53, and CloudFront** for improved reliability, speed, and global availability. A custom domain will also be configured for professional deployment.

Conclusion

The ArtCraft E-commerce Platform marks the successful completion of a comprehensive full-stack development journey, combining theory, practical implementation, and real-world problem-solving. The final outcome demonstrates a functional, user-friendly solution that fulfills all core objectives despite time constraints and academic schedules. Completing the essential modules—user authentication, product management, cart and checkout, order workflows, admin dashboard, and mobile responsiveness—reflects the team's commitment to delivering a solid and reliable platform.

Throughout the development process, the team gained significant experience in the full-stack workflow, particularly understanding how frontend and backend components interact, how APIs are designed and consumed, and how deployment in real hosting environments works. React, Node.js, and deployment practices initially posed challenges, but overcoming them contributed to strong personal growth and technical improvement. Collaboration, task-sharing, and persistence helped the team maintain focus even during placement season, and the successful placement of all team members adds pride and motivation to this achievement.

The platform provides practical value by offering an affordable, simple, open-source solution that benefits store owners, students, and general customers seeking art and craft supplies. Its clean interface, ease of use, and learning-oriented design make it accessible for small businesses and learners who may not have access to expensive software tools. Being open source further strengthens its usefulness by allowing others to study, customize, and build upon the project.
Looking ahead, the system has strong potential to evolve into a production-ready platform with added features such as online payments, enhanced analytics, mobile apps, improved profile management, and cloud-based deployment on AWS with a custom domain. This project forms a

solid foundation for future learning, experimentation, and real-world applications, reflecting both the technical competence and dedication developed throughout its creation.

Bibliography & References

This section lists all documentation sources, learning materials, and technical references used throughout the development of the ArtCraft E-commerce Platform.

1. React Official Documentation
   ReactJS. “React Documentation.” Meta Platforms, Inc.
   https://react.dev
2. Node.js Official Documentation
   Node.js Foundation. “Node.js Documentation.”
   https://nodejs.org
3. Express.js Official Documentation
   Express.js Team. “Express Framework Documentation.”
   https://expressjs.com
4. Mongoose / MongoDB ODM Documentation
   Mongoose Developers. “Mongoose Documentation.”
   https://mongoosejs.com
5. MongoDB Atlas Documentation
   MongoDB Inc. “MongoDB Atlas Documentation.”
   https://www.mongodb.com/atlas
6. JSON Web Token JWT Documentation
   Auth0. “JSON Web Tokens Introduction & Usage.”
   https://jwt.io
7. Tailwind CSS Documentation
   Tailwind Labs. “Tailwind CSS Documentation.”
   https://tailwindcss.com
8. Vite Documentation
   Vite Contributors. “Vite Official Documentation.”
   https://vitejs.dev
9. MDN Web Docs

Mozilla Foundation. “MDN Web Docs – HTML, CSS, JavaScript, Web APIs.”
https://developer.mozilla.org

10. W3Schools Web Development Tutorials
W3schools.com. “HTML, CSS, JS and Web Development Tutorials.”
https://www.w3schools.com

11. Medium Blogs General Web Development Articles
Medium.com. “Web Development Blogs and Articles.”
https://medium.com

12. GitHub Documentation
GitHub Inc. “GitHub Documentation & Version Control Guide.”
https://docs.github.com

13. Render.com Deployment Documentation
Render. “Render Cloud Hosting Documentation.”
https://render.com/docs

14. Google Gemini API Documentation
Google AI. “Gemini API Developer Documentation.”
https://ai.google.dev/gemini-api

15. Postman API Platform Documentation
Postman. “Postman Documentation.”
https://learning.postman.com

16. Visual Studio Code Documentation
Microsoft. “VS Code Documentation.”
https://code.visualstudio.com/docs

17. ChatGPT OpenAI
OpenAI. “ChatGPT Assistant Usage for Code Explanations and JWT Concepts.”
https://openai.com

18. DeepSeek AI Documentation
DeepSeek. “DeepSeek Code & Assistant Documentation.”
https://deepseek.com

19. Claude AI Documentation
Anthropic. “Claude AI Documentation & Usage.”
https://claude.ai