SMART E-COMMERCE PRICE TRACKING

Training Report Submitted in Partial Fulfilment of the Requirements for the Degree of

Bachelor of Engineering
in
Computer Science and Engineering

Submitted by

Mohit Kumar Jindal: (Roll No. 22UCSE4024)

Arpit Singhal:(Roll No. 2UCSE4006)

Vinay Gupta:(Roll No. 22UCSE4039)

Rahul Singh:(Roll No. 22UCSE4029)

Under the Supervision of

Dr. Shrawn Balach
Head Of Department
MBM University, Jodhpur

Department of Computer Science and Engineering
MBM University, Jodhpur
May, 2025

SMART E-COMMERCE PRICE TRACKING

Training Report Submitted in Partial Fulfilment of the Requirements for the Degree of

Bachelor of Engineering
in
Computer Science and Engineering

Submitted by

Mohit Kumar Jindal: (Roll No. 22UCSE4024)

Arpit Singhal:(Roll No. 2UCSE4006)

Vinay Gupta:(Roll No. 22UCSE4039)

Rahul Singh:(Roll No. 22UCSE4029)

Under the Supervision of

Dr. Shrawn Balach
Head Of Department
MBM University, Jodhpur

Department of Computer Science and Engineering
MBM University, Jodhpur
May, 2025

DECLARATION

We,**Mohit Kumar Jindal** hereby declare that this work titled **SMART E-COMMERCE PRICE TRACKING** is a record of original work done by me under the supervision and guidance of **Dr. Shrawn Balach,Head Of Department** at **MBM University,Jodhpur** from Aug 2024 to May 2025.

We, further certify that this work has not formed the basis for the award of the Degree/Diploma/Associateship/Fellowship or similar recognition to any candidate of any university and no part of this report is reproduced as it is from any other source without appropriate reference and permission.

Sign Of Student
Mohit Kumar Jindal
(Roll No. 22UCSE4024)

Sign Of Student
Arpit Singhal
(Roll No. 2UCSE4006)

Sign Of Student
Vinay Gupta
(Roll No. 22UCSE4039)

Sign Of Student
Rahul Singh
(Roll No. 22UCSE4029)

ACKNOWLEDGEMENT

We take this opportunity to express our sincere gratitude to all those who have extended their valuable support, guidance, and encouragement throughout the course of our final year project, **"Smart E-Commerce Price Tracking."**

First and foremost, we would like to extend our heartfelt thanks to our project supervisor, **Dr. Shrawn Balach**, Professor, Department of Computer Science and Engineering, MBM University, Jodhpur, for his unwavering support, insightful guidance, and constant motivation. His valuable suggestions and deep knowledge in the domain of modern web technologies and software engineering played a critical role in the successful completion of this project.

We are also thankful to the faculty members and technical staff of the **Department of Computer Science and Engineering**, MBM University, for providing a conducive learning environment and their invaluable feedback throughout the development cycle.

Our sincere thanks to our institution, **MBM University, Jodhpur**, for facilitating the resources and infrastructure that were instrumental in completing this project successfully.

We would also like to acknowledge the collaborative spirit and commitment of each member of our project team:

- **Rahul Singh**, for leading the web scraping module and managing cron automation using Puppeteer, Cheerio, and Bright Data.
- **Vinay Gupta**, for building the robust backend services and integrating the MongoDB database.
- **Mohit Kumar Jindal**, for designing and implementing the responsive frontend using Next.js 15 and Tailwind CSS.
- **Arpit Singhal**, for developing the admin dashboard and analytics module for administrative control.

ABSTRACT

The exponential growth of e-commerce over the past decade has significantly reshaped consumer behavior and the retail landscape. With thousands of products listed across platforms such as Amazon, Flipkart, and eBay, customers face challenges in tracking fluctuating prices and stock availability. To address this problem, our project, titled **"Smart E-Commerce Price Tracking,"** proposes an automated, scalable solution that utilizes advanced web technologies to provide real-time product monitoring and intelligent notifications.

This system is implemented as a full-stack web application built using **Next.js 15** for the frontend, **Node.js** and **Express.js** for the backend, and **MongoDB** for persistent data storage. It leverages a powerful web scraping pipeline designed using **Bright Data Web Unlocker**, **Puppeteer**, and **Cheerio**, which enables the system to extract dynamic content from e-commerce platforms while bypassing bot detection mechanisms and CAPTCHAs (Bright Data, 2023).

Key features of the system include secure user authentication via email or Google OAuth 2.0, a user-friendly dashboard for tracking multiple product URLs, email notifications for price drops or stock status changes, and an admin interface with tools for monitoring scraping operations and analyzing user engagement. The scraping process is fully automated through the use of **cron jobs**, which run at scheduled intervals to fetch and update product data in real time.

This solution not only improves consumer purchasing decisions by delivering timely alerts, but also serves as a valuable tool for e-commerce sellers who can track competitors' stock availability and adjust pricing strategies accordingly (Gandomi & Haider, 2015). Moreover, the modular and scalable architecture ensures easy integration of future enhancements such as AI-driven price forecasting, multi-platform support, and mobile or voice assistant applications.

In conclusion, this project demonstrates the practical integration of modern technologies in solving real-world e-commerce problems through intelligent automation, responsive design, and secure data practices.

Table of Contents

List of Figures

Chapter 1 INTRODUCTION

1.1 Background of E-commerce and Dynamic Pricing

1.1.1 Evolution of E-commerce

The landscape of commerce has undergone a profound transformation over the past few decades, evolving from traditional brick-and-mortar establishments to a pervasive digital marketplace. The advent of the internet in the late 20th century laid the foundational groundwork, but it was the widespread adoption of broadband, secure online payment gateways, and sophisticated logistics networks that truly catalyzed the e-commerce revolution. Initially, online retail was characterized by static product listings and fixed pricing, largely mirroring conventional retail models. However, as technology advanced and competition intensified, e-commerce platforms began to leverage data analytics and real-time market dynamics to optimize their strategies. This evolution saw the emergence of massive online retailers like Amazon, eBay, and later, specialized platforms, fundamentally altering consumer purchasing habits and business operations globally. The convenience of 24/7 access, vast product selections, and direct-to-consumer models have made e-commerce an indispensable part of modern life, pushing businesses to constantly innovate in how they present products and manage pricing

1.1.2 Understanding Dynamic Pricing Strategies

Dynamic pricing, also known as surge pricing, demand pricing, or time-based pricing, is a strategy in which businesses set flexible prices for products or services based on current market demands, supply levels, competitor pricing, time of day, customer behavior, and other external factors. Unlike traditional fixed pricing, dynamic pricing algorithms continuously adjust prices in real-time to maximize revenue and profit. For instance, airline tickets and hotel rooms are classic examples where prices fluctuate based on booking time, demand, and availability. In the e-commerce sector, this strategy is employed by retailers to respond to competitor price changes, inventory levels, promotional events, and even individual user browsing history. The algorithms behind dynamic pricing are often complex, incorporating machine learning models that analyze vast datasets to predict optimal price points. While beneficial for businesses in optimizing sales and inventory, this fluidity in pricing can create a significant challenge for consumers who seek to purchase products at the most opportune moment.

1.1.3 Consumer Challenges in Online Shopping

The dynamic nature of e-commerce, while offering unparalleled convenience and choice, presents several significant challenges for consumers. Foremost among these is the difficulty in tracking price fluctuations. A product viewed in the morning might have a different price by the afternoon, leading to consumer frustration and the potential for overpaying. Without a systematic way to monitor these changes, consumers often resort to manual checks, which are time-consuming, inefficient, and prone to missing optimal buying windows. Furthermore, stock availability is another critical concern. Popular items can go out of stock rapidly, and consumers often miss restock notifications, leading to missed purchase opportunities.

1.2 Problem Statement

1.2.1 The Need for Price Monitoring

In today's highly competitive and rapidly evolving e-commerce landscape, the ability to effectively monitor product prices is no longer a luxury but a fundamental necessity for both consumers and businesses. For consumers, the primary need stems from the desire to make financially prudent purchasing decisions. With prices constantly shifting due to dynamic pricing strategies, seasonal sales, flash deals, and competitor actions, manually checking product prices across various online stores is an impractical and often futile endeavor. Consumers require a reliable mechanism to identify price drops, ensuring they can acquire desired items at their lowest possible cost. This need is particularly acute for high-value purchases where even a small percentage drop can translate into significant savings. Beyond individual savings, effective price monitoring fosters greater transparency in the market, empowering consumers to navigate the complexities of online retail with confidence. For businesses, particularly smaller retailers or those operating in niche markets, monitoring competitor pricing is crucial for maintaining a competitive edge and adjusting their own strategies in real-time.

1.2.2 Limitations of Manual Tracking

The traditional approach to price monitoring, which involves manually visiting multiple e-commerce websites repeatedly, is fraught with severe limitations. Firstly, it is incredibly time-consuming and inefficient. An individual might spend hours refreshing product pages, only to miss a fleeting price drop that occurs outside their monitoring window. This manual effort scales poorly; tracking more than a handful of products becomes an unmanageable task. Secondly, human error is a significant factor; it is easy to overlook subtle price changes or misrecord data, leading to inaccurate assessments. Thirdly, manual tracking cannot keep pace with the speed of dynamic pricing algorithms. By the time a consumer manually checks a price, it might have already changed multiple times. This reactive approach leaves consumers at a distinct disadvantage, often leading to missed opportunities for significant savings or stock availability alerts. Furthermore, manual methods offer no historical data analysis, preventing consumers from understanding long-term price trends or identifying optimal buying patterns. These inherent limitations highlight the critical need for an automated, real-time solution.

1.2.3 Challenges for E-commerce Sellers

While consumers face challenges in finding the best deals, e-commerce sellers encounter their own set of difficulties in the dynamic online marketplace. A key challenge for sellers is maintaining competitive pricing. Without real-time insights into competitor pricing and stock levels, businesses risk overpricing their products and losing sales, or underpricing and sacrificing potential revenue. Manual competitor analysis is as inefficient for sellers as manual price tracking is for consumers. Furthermore, understanding market demand and supply dynamics is crucial for inventory management. If a competitor's product goes out of stock, it presents an opportunity for other sellers to adjust their pricing or increase their marketing efforts for similar items. Conversely, a sudden influx of supply from competitors can necessitate price adjustments to remain competitive. Beyond pricing, sellers also need to monitor product availability to prevent stockouts or overstocking, which can lead to lost sales or increased holding costs. The absence of automated, real-time competitive intelligence can severely hinder a seller's ability to react swiftly to market changes, impacting their profitability and market share.

1.3 Proposed Solution: Smart E-Commerce Price Tracking

1.3.1 Overview of the System

The proposed "Smart E-Commerce Price Tracking" system is designed as a robust, full-stack web application aimed at revolutionizing how consumers and sellers interact with dynamic online prices. Built upon a modern technology stack comprising Next.js 15 for the frontend, Node.js with Express.js for the backend, and MongoDB for data persistence, the system provides a comprehensive solution for automated price and stock monitoring. At its core, the system leverages advanced web scraping techniques, utilizing Bright Data Web Unlocker, Puppeteer, and Cheerio, to extract real-time product information from various e-commerce platforms. This extracted data is then processed, analyzed for changes, and stored in a structured database. Automated cron jobs are configured to periodically trigger these scraping tasks, ensuring that the price and stock information remains consistently up-to-date. The system also integrates a sophisticated notification engine that dispatches real-time alerts to users via email when a tracked product experiences a price drop or becomes available in stock. Furthermore, a dedicated admin dashboard provides powerful tools for managing users, products, and monitoring the system's operational health, ensuring a seamless and efficient experience for all stakeholders.

1.3.2 Key Features and Capabilities

The Smart E-Commerce Price Tracking system boasts a rich set of features designed to provide a seamless and powerful user experiencSecure User Authentication: Users can securely register and log in using email/password or conveniently via Google authentication, ensuring personalized tracking experiences.

Intuitive Product Tracking: A user-friendly interface allows users to easily input product URLs from supported e-commerce sites. The system then takes over, automatically monitoring these products.

Real-time Price Drop Alerts: Automated email notifications are dispatched instantly when the price of a tracked product falls below a user-defined threshold or simply drops from its previous recorded price.

Stock Availability Notifications: Users receive timely alerts when a previously out-of-stock product becomes available for purchase, preventing missed opportunities.

Personalized Wishlist & Favorites: Users can save and organize their desired products in a dedicated wishlist, making it easy to manage and track multiple items of interest.Comprehensive Admin Dashboard: Administrators gain full control over the system, with modules for user management (view, edit, activate/deactivate accounts), product management (monitor all tracked products, manually adjust details), and a scraping control panel to manage job frequencies and monitor performance.

Data Analytics & Reporting: The admin dashboard provides valuable insights, including historical price trend visualizations, user engagement metrics, and notification effectiveness reports.

Robust Web Scraping Engine: The backend employs a sophisticated scraping engine capable of navigating complex website structures, handling dynamic content, and bypassing common anti-scraping measures using Bright Data Web Unlocker.

Automated Data Refresh: Scheduled cron jobs ensure that product data is consistently updated at predefined intervals, guaranteeing the freshness and accuracy of information.

Scalable Architecture: Designed with scalability in mind, the system can handle a growing number of users and tracked products without significant performance degradation.

1.4 Project Objectives

1.4.1 Primary Objectives

The primary objectives of the Smart E-Commerce Price Tracking project are foundational to its success and address the core problems identified:

Develop a robust and scalable web scraping engine: The paramount objective is to build a highly reliable web scraping component capable of extracting accurate and real-time price and stock availability data from various e-commerce websites. This includes overcoming common anti-scraping mechanisms such as CAPTCHAs, IP blocking, and dynamic content loading, ensuring a high success rate and data integrity.

Implement an efficient and secure user authentication system: To provide personalized tracking services, a secure and user-friendly authentication system must be developed. This system will support both traditional email/password registration and modern social login options (e.g., Google), ensuring user data privacy and system integrity through robust security protocols like JWT.

Design and implement a real-time notification system: A critical objective is to create an automated notification system that promptly alerts users via email about significant price drops or changes in product stock availability. This system must be reliable, timely, and configurable to user preferences, ensuring that users receive actionable information when it matters most.

1.4.2 Secondary Objectives

Beyond the core functionalities, several secondary objectives aim to enhance the system's value, usability, and future potential:

Integrate historical price data visualization: To provide users with deeper insights, a secondary objective is to implement interactive charts and graphs that visualize the historical price trends of tracked products. This will enable users to identify patterns and make more informed purchasing decisions based on past data.
Ensure system performance and optimization: The system must be optimized for speed and efficiency, particularly concerning scraping operations and notification delivery. This includes optimizing database queries, ensuring quick response times for API calls, and minimizing resource consumption to support a growing user base.

Implement robust error handling and logging for scraping operations: Given the inherent challenges of web scraping, a crucial secondary objective is to build comprehensive error handling mechanisms and detailed logging for all scraping jobs. This will allow for quick identification and resolution of issues, ensuring the continuous and reliable operation of the scraping engine.

Develop a scalable and maintainable codebase: Adhering to best practices in software development, a secondary objective is to write clean, modular, and well-documented code that facilitates future enhancements and maintenance. The architecture should be designed to accommodate future growth in features and user volume.

Prepare for future expansion to multiple e-commerce platforms: While initial focus may be on Amazon, a secondary objective is to design the scraping engine with an extensible architecture that can easily integrate support for additional e-commerce platforms (e.g., Flipkart, eBay, Myntra) in the future, broadening the system's utility.

Chapter 2 TECHNOLOGY USED

2.1 Frontend Technologies

2.1.1 Next.js 15

2.1.1.1 Advantages for Modern Web Applications

Next.js 15 stands as a leading React framework, offering a robust and highly performant foundation for building modern web applications. Its primary advantage lies in its hybrid rendering capabilities, supporting Server-Side Rendering (SSR), Static Site Generation (SSG), and Incremental Static Regeneration (ISR). This flexibility allows developers to choose the optimal rendering strategy for each page, leading to significant improvements in initial load times (Time to First Byte) and overall Search Engine Optimization (SEO). Unlike traditional client-side React applications that suffer from empty initial HTML, Next.js pre-renders content, making it immediately visible to search engines and users. Furthermore, Next.js provides built-in API routes, enabling developers to create backend endpoints directly within the Next.js project, simplifying full-stack development and reducing context switching. Its automatic code splitting ensures that only the necessary JavaScript is loaded for each page, minimizing bundle sizes and accelerating page navigation.

2.1.1.2 Key Features Utilized (e.g., Server Components, Routing)

In the Smart E-Commerce Price Tracking project, Next.js 15's advanced features were leveraged extensively to build a highly performant and maintainable user interface.
Server Components: This revolutionary feature was crucial for optimizing data fetching and rendering performance. By rendering components on the server, we minimized the amount of JavaScript sent to the client, leading to faster initial page loads and improved responsiveness. For instance, the product listing page and individual product detail pages, which display frequently updated price data, benefited immensely from Server Components fetching data directly from the backend API, reducing client-side data fetching overhead.
Data Fetching Methods: We utilized getServerSideProps for pages requiring fresh data on every request (like the current price display on the dashboard) and getStaticProps with revalidate for content that could be periodically updated (like product descriptions that don't change as frequently as prices). This hybrid approach ensured optimal performance and data freshness.
API Routes: Next.js API routes were used to create lightweight backend endpoints for specific functionalities, such as handling form submissions or simple data queries, directly within the frontend repository. While the main backend logic resides in a separate Node.js/Express.js application, API routes provided a convenient way to integrate certain client-side interactions with server-side logic without deploying a separate microservice for every small task.

2.1.2 Tailwind CSS

2.1.2.1 Utility-First CSS Framework Benefits

Tailwind CSS is a highly customizable, utility-first CSS framework that provides low-level utility classes to build designs directly in your markup. Its primary benefit lies in accelerating the development of user interfaces by eliminating the need to write custom CSS from scratch for every component. Instead of defining semantic class names (e.g., .card-header), developers apply pre-defined utility classes (e.g., bg-white p-6 rounded-lg shadow-md) directly to HTML elements. This approach leads to highly consistent designs, as the same utility classes are reused across the application, reducing the chances of design inconsistencies. Furthermore, Tailwind CSS significantly minimizes CSS file sizes by only including the utilities actually used in the project, which is particularly beneficial for production deployments and faster page loads. The "utility-first" paradigm also encourages a component-based design philosophy, where complex UI elements are composed from smaller, reusable utility classes, leading to more modular and maintainable code. This framework’s flexibility allows for rapid prototyping and iterative design, making it ideal for a project like the Smart E-Commerce Price Tracker where a clean, responsive, and modern aesthetic is crucial.

2.1.2.2 Responsive Design Implementation

Tailwind CSS excels in facilitating responsive design, ensuring that the Smart E-Commerce Price Tracking application provides an optimal viewing and interaction experience across a wide range of devices, from mobile phones to large desktop monitors. Tailwind's responsive prefixes (e.g., sm:, md:, lg:, xl:) allow for conditional styling based on predefined breakpoints. For instance, a layout might use flex flex-col md:flex-row to stack elements vertically on small screens and arrange them horizontally on medium screens and above. This granular control over responsiveness means that elements like navigation bars, product cards, and data tables fluidly adapt to different screen sizes, preventing horizontal scrolling and ensuring readability. We extensively utilized these prefixes for adjusting padding (p-4 sm:p-6), margin (mx-2 md:mx-4), font sizes (text-base lg:text-lg), and even element visibility (hidden md:block). The grid and flexbox utilities (grid, flex, gap-4) were instrumental in creating adaptive layouts for the product dashboard and admin panels, ensuring that content is always well-organized and accessible regardless of the viewport. This approach allowed for a highly tailored responsive experience without writing any custom media queries, significantly streamlining the development and maintenance of the UI across various device form factors.

2.2 Backend Technologies

2.2.1 Node.js

2.2.1.1 Asynchronous, Event-Driven Architecture

Node.js serves as the backbone of the Smart E-Commerce Price Tracking system's backend, primarily due to its asynchronous, event-driven architecture. Built on Chrome's V8 JavaScript engine, Node.js allows for non-blocking I/O operations, meaning that it can handle multiple requests concurrently without waiting for previous operations (like database queries or external API calls) to complete. This is achieved through an event loop mechanism: when an I/O operation is initiated, Node.js registers a callback and moves on to process other requests. Once the I/O operation finishes, the callback is pushed to the event queue and executed. This model makes Node.js exceptionally efficient for I/O-bound tasks, which are prevalent in our application (e.g., fetching data from MongoDB, making HTTP requests to e-commerce sites for scraping, sending email notifications). Unlike traditional thread-based servers, Node.js uses a single-threaded event loop, which simplifies concurrency management and reduces overhead, making it highly suitable for building scalable network applications. This architecture is crucial for a price tracking system that needs to perform numerous concurrent scraping operations and handle real-time notifications without becoming a bottleneck.

2.2.1.2 Scalability and Performance

The choice of Node.js for the backend is also heavily influenced by its inherent scalability and high performance, critical attributes for an application that anticipates a growing user base and increasing demand for real-time data. Node.js's non-blocking I/O model allows it to handle a large number of concurrent connections with minimal overhead, making it highly efficient for applications that require rapid data processing and delivery. For the Smart E-Commerce Price Tracking system, this translates to faster scraping execution, quicker processing of price change detections, and near-instantaneous notification delivery. The lightweight nature of Node.js, combined with its ability to leverage JavaScript across both frontend and backend, facilitates faster development cycles and easier maintenance. Furthermore, the Node.js ecosystem, with its vast npm registry, provides a plethora of optimized libraries and tools that can be readily integrated to enhance performance, such as caching mechanisms, load balancing solutions, and efficient database drivers. Its ability to scale horizontally by running multiple instances across different servers also ensures that the application can handle increasing traffic and data processing demands without compromising on responsiveness or reliability, making it a robust choice for a dynamic and data-intensive application.

2.2.2 Express.js

2.2.2.1 RESTful API Development

Express.js, a minimalist and flexible Node.js web application framework, is the chosen tool for building the RESTful APIs that power the Smart E-Commerce Price Tracking system. Express simplifies the process of defining routes, handling HTTP requests (GET, POST, PUT, DELETE), and sending responses. It provides a robust routing system that allows us to structure our API endpoints logically, such as /api/users for user management, /api/products for tracked products, and /api/notifications for notification-related actions. This adherence to RESTful principles ensures a clear, stateless, and scalable communication interface between the frontend, backend, and various internal services. For instance, a POST /api/products request would be used to add a new product URL for tracking, while a GET /api/products/:id would retrieve details for a specific tracked item, including its price history. Express's middleware architecture also allows for easy integration of functionalities like body parsing, authentication checks (e.g., JWT validation), and error handling, ensuring that every API request is processed securely and efficiently. This structured approach to API development is crucial for maintaining a clean separation of concerns and facilitating future integrations or modifications.

2.2.2.2 Middleware and Routing

Express.js's strength lies significantly in its powerful middleware and routing capabilities, which are fundamental to the structured and secure operation of the Smart E-Commerce Price Tracking backend.

Routing: Express provides an intuitive way to define routes based on HTTP methods and URL paths. For example, app.get('/api/products', ...) handles requests to fetch all tracked products, while app.post('/api/products', ...) handles requests to add a new product. This allows for clear organization of API endpoints, making the backend logic easy to understand and maintain. We utilize dynamic routing (e.g., app.get('/api/products/:id', ...)) to fetch specific product details, where :id is a placeholder for the product's unique identifier.

Middleware: Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application’s request-response cycle. They can execute code, make changes to the request and response objects, end the request-response cycle, and call the next middleware in the stack. In our project, key middleware functions include:

Body Parsers: express.json() and express.urlencoded() are used to parse incoming request bodies in JSON and URL-encoded formats, respectively, making it easy to access data sent from the frontend.

Authentication Middleware: A custom middleware function is implemented to validate JWTs sent with requests. This middleware checks the token's validity, extracts user information, and attaches it to the req object, ensuring that only authenticated and authorized users can access protected routes (e.g., adding products, viewing personal dashboards).

Error Handling Middleware: A centralized error handling middleware catches any exceptions thrown during the request-response cycle, logs them, and sends a standardized error response to the client, improving the robustness and user experience of the API.

CORS (Cross-Origin Resource Sharing) Middleware: To allow the frontend application (which might be hosted on a different domain or port) to make requests to the backend API, CORS middleware is configured to enable secure cross-origin communication.
This modular approach with middleware allows for a clean separation of concerns, making the codebase more organized, testable, and scalable.

2.2.3 MongoDB (Database)

2.2.3.1 NoSQL Document Database Advantages

MongoDB is chosen as the primary database for the Smart E-Commerce Price Tracking system due to its NoSQL, document-oriented nature, which offers significant advantages for handling the flexible and evolving data structures inherent in e-commerce product information. Unlike traditional relational databases, MongoDB stores data in BSON (Binary JSON) documents, allowing for dynamic schemas. This flexibility is crucial because product data from various e-commerce sites might have differing attributes (e.g., some products might have "color" and "size" variations, while others might not). MongoDB's schema-less nature allows us to store diverse product information without rigid predefined tables, simplifying data modeling and enabling faster iteration during development. Furthermore, MongoDB's horizontal scalability through sharding allows it to distribute data across multiple servers, making it highly suitable for handling large volumes of data and high read/write loads as the number of tracked products and users grows. Its native JSON-like document structure also aligns perfectly with JavaScript and Node.js, reducing the need for complex object-relational mapping and streamlining data persistence operations.

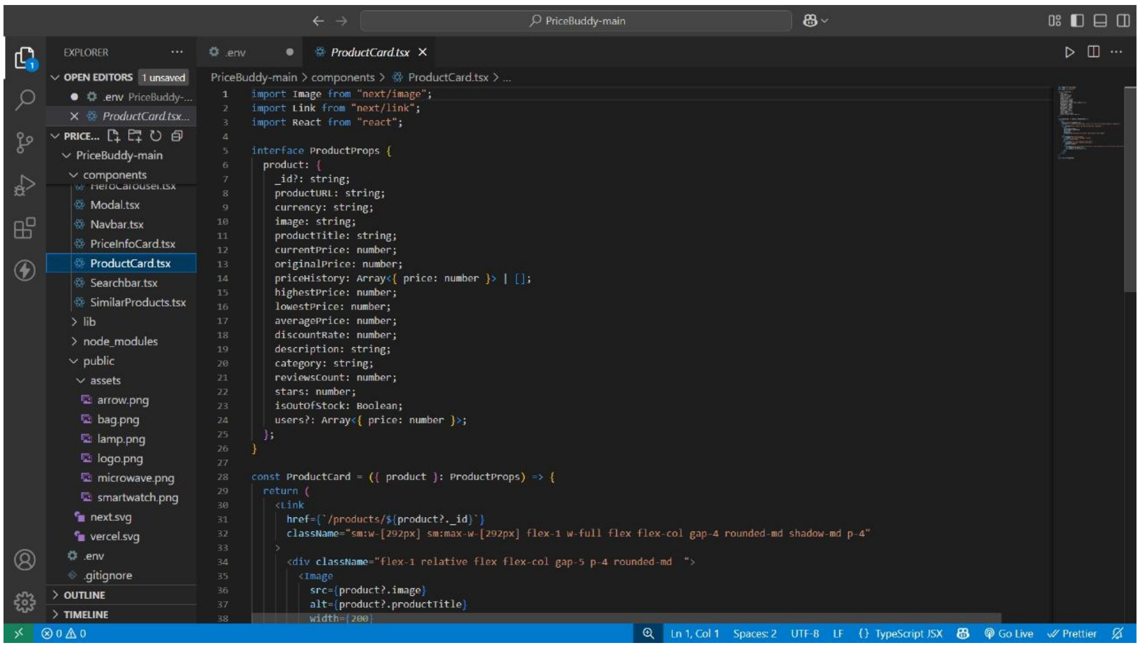
2.2.3.2 Data Modeling for Price Tracking

Effective data modeling in MongoDB is crucial for efficient storage and retrieval of price tracking information. We design several key collections to store the application's data:

users Collection: Stores user authentication details (email, hashed password, Google ID), user preferences (e.g., notification settings), and a list of trackedProductIds that each user is monitoring. This collection is central to user-specific data.

```
{
  "_id": ObjectId("..."),
  "email": "user@example.com",
  "passwordHash": "...",
  "googleId": "...",
  "notificationSettings": {
    "priceDropEmails": true,
    "stockAvailabilityEmails": true
  },
  "trackedProductIds": [
    ObjectId("product1_id"),
    ObjectId("product2_id")
  ],
  "createdAt": ISODate("..."),
  "updatedAt": ISODate("...")
}
```

products Collection: This is the core collection for storing information about each unique product being tracked. Each document includes static product details and an embedded array for price history.


```tsx
import Image from "next/image";
import Link from "next/link";
import React from "react";

interface ProductProps {
  product: {
    _id?: string;
    productURL: string;
    currency: string;
    image: string;
    productTitle: string;
    currentPrice: number;
    originalPrice: number;
    priceHistory: Array<{ price: number }> | [];
    highestPrice: number;
    lowestPrice: number;
    averagePrice: number;
    discountRate: number;
    description: string;
    category: string;
    reviewsCount: number;
    stars: number;
    isOutOfStock: Boolean;
    users?: Array<{ price: number }>;
  };
}

const ProductCard = ({ product }: ProductProps) => {
  return (
    <Link
      href={`/products/${product?._id}`}
      className="sm:w-[292px] sm:max-w-[292px] flex-1 w-full flex flex-col gap-4 rounded-md shadow-md p-4"
    >
      <div className="flex-1 relative flex flex-col gap-5 p-4 rounded-md  ">
        <Image
          src={product?.image}
          alt={product?.productTitle}
          width={200}
```

Figure 1 Price tracking Model

This document-oriented approach allows for efficient querying of product information and its associated history, while also providing the flexibility to adapt to new data points as the system evolves. Indexing on frequently queried fields like url, userId, productId, and timestamp within priceHistory will ensure optimal read performance.

2.3 Web Scraping Technologies

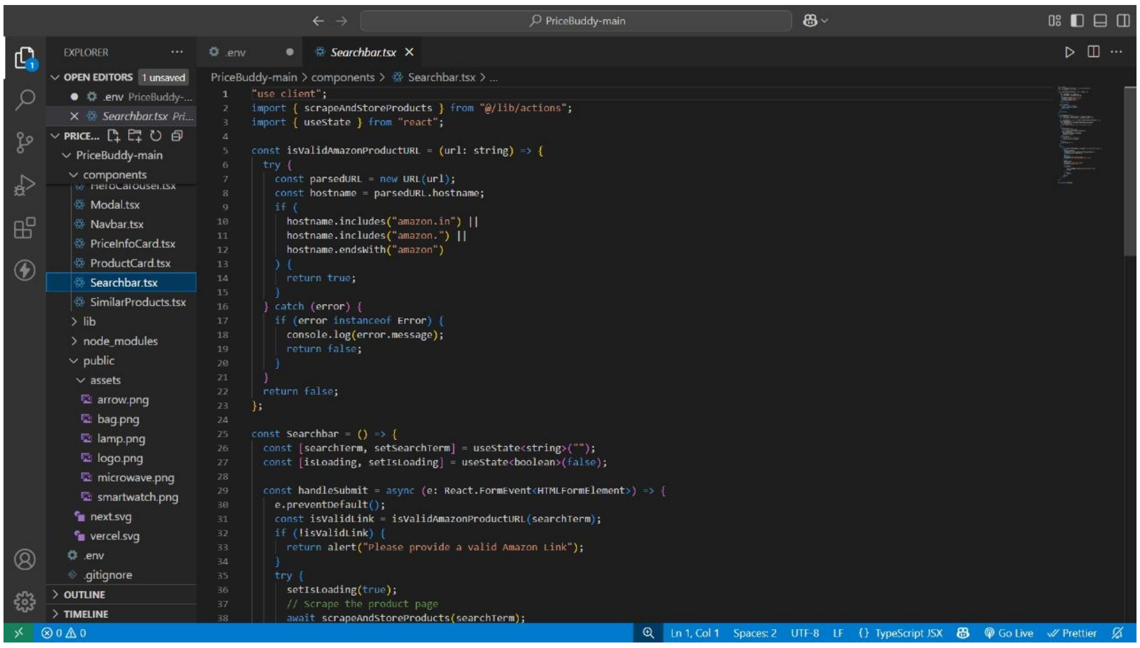
2.3.1 Bright Data Web Unlocker

2.3.1.1 Overcoming Anti-Scraping Measures

Web scraping, while powerful, often faces significant hurdles from anti-scraping measures implemented by websites. These measures include CAPTCHAs, IP blocking, dynamic content rendering, browser fingerprinting detection, and complex JavaScript challenges. Bright Data Web Unlocker is a crucial component in our scraping architecture precisely because it intelligently bypasses these sophisticated defenses. Instead of directly sending requests from our server, we route our scraping requests through Bright Data's network. The Web Unlocker automatically handles retries, IP rotation, CAPTCHA solving, browser emulation, and JavaScript rendering, presenting itself as a legitimate user. It uses a vast pool of residential and mobile IPs, making it extremely difficult for target websites to identify and block our scraping activity. This service acts as an intelligent proxy layer, allowing our scraping engine to focus solely on data extraction logic rather than spending significant development effort on bypassing anti-bot systems. This strategic integration ensures a high success rate for data retrieval, even from highly protected e-commerce sites, guaranteeing the reliability and freshness of the price and stock information.

2.3.1.2 Integration with Scraping Logic

Integrating Bright Data Web Unlocker into our scraping logic is a streamlined process that significantly enhances the reliability of data extraction. Our Node.js backend sends HTTP requests (or initiates Puppeteer browser sessions) through the Web Unlocker's proxy endpoint. Instead of making a direct fetch or axios call to the target e-commerce URL, we configure our requests to go through the Bright Data proxy. The Web Unlocker then handles the complexities of routing, IP rotation, and anti-bot bypass before forwarding the clean HTML content back to our scraping engine. For Puppeteer, this involves configuring the browser launch arguments to use the Bright Data proxy.



```
"use client";
import { scrapeAndStoreProducts } from "@/lib/actions";
import { useState } from "react";

const isValidAmazonProductURL = (url: string) => {
  try {
    const parsedURL = new URL(url);
    const hostname = parsedURL.hostname;
    if (
      hostname.includes("amazon.in") ||
      hostname.includes("amazon.") ||
      hostname.endsWith("amazon")
    ) {
      return true;
    }
  } catch (error) {
    if (error instanceof Error) {
      console.log(error.message);
      return false;
    }
  }
  return false;
};

const Searchbar = () => {
  const [searchTerm, setSearchTerm] = useState<string>("");
  const [isLoading, setIsLoading] = useState<boolean>(false);

  const handleSubmit = async (e: React.FormEvent<HTMLFormElement>) => {
    e.preventDefault();
    const isValidLink = isValidAmazonProductURL(searchTerm);
    if (!isValidLink) {
      return alert("Please provide a valid Amazon Link");
    }
    try {
      setIsLoading(true);
      // Scrape the product page
      await scrapeAndStoreProducts(searchTerm);
```

Figure 2 *Integration with Scraping Logic*

This seamless integration means that our core scraping scripts (using Puppeteer for dynamic content and Cheerio for static parsing) can operate as if they are accessing the websites directly, while Bright Data handles all the underlying network and anti-bot complexities. This modularity allows us to maintain clean, focused scraping code and ensures consistent data acquisition.

2.3.1.3 Data Extraction Techniques

Leveraging Cheerio's jQuery-like API, we employ various data extraction techniques to accurately pull relevant information from e-commerce product pages.

Selector-Based Extraction: The most common technique involves identifying unique CSS selectors for elements containing the desired data. For instance, to extract a product's title, we might use $('.product-title-class').text(). For the price, it could be $('#priceblock_ourprice').text() or a more complex selector like $('div[data-component-type="price-display"] span.price').text().

Attribute Extraction: Beyond text content, we often need to extract attributes from HTML elements. For example, $('img.product-image').attr('src') would retrieve the URL

of the product image. Similarly, $('a.product-link').attr('href') would get the canonical product URL.
Iterating Over Elements: For lists of items (e.g., product variations, reviews), Cheerio allows us to iterate using .each():

```
$('.product-variations li').each((index, element) => {
  const variationName = $(element).find('.variation-name').text();
  const variationPrice = $(element).find('.variation-price').text();
  // Store variation data
});
```

Regular Expressions (with text content): In cases where the desired data is embedded within a larger text string or doesn't have a distinct HTML element, we can extract the text content using Cheerio and then apply regular expressions to parse out the specific value (e.g., extracting a numerical price from a string like "Price: $99.99 USD").
Handling Missing Elements: Robust scraping involves checking if an element exists before attempting to extract its content, using if ($(selector).length > 0). This prevents errors when a website's structure varies or an element is occasionally missing.
By combining these techniques, Cheerio enables precise and efficient data extraction, transforming raw HTML into structured product information ready for database storage and analysis.

2.4 Automation and Deployment

2.4.1 Cron Jobs

2.4.1.1 Scheduling Periodic Tasks

Cron jobs are an essential component of the Smart E-Commerce Price Tracking system, serving as the backbone for automating periodic tasks, most notably the continuous web scraping process. A cron job is a time-based job scheduler in Unix-like computer operating systems. Users who set up and maintain software environments use cron to schedule jobs (commands or shell scripts) to run periodically at fixed times, dates, or intervals. In our context, cron jobs are configured to trigger the web scraping engine at predefined intervals (e.g., every hour, every few hours, or daily, depending on the product and desired freshness). This ensures that product prices and stock availability are consistently updated without manual intervention. The scheduling mechanism allows for fine-grained control over when and how frequently scraping occurs, optimizing resource usage and minimizing the load on target e-commerce websites. For instance, high-demand products might be scraped more frequently (e.g., every 30 minutes), while less volatile items could be updated less often (e.g., once a day). This automation is critical for maintaining the "real-time" aspect of the price tracking system and ensuring that users receive timely and accurate notifications.

2.4.1.2 Ensuring Data Freshness

The primary role of automated cron jobs in the Smart E-Commerce Price Tracking system is to ensure the freshness and accuracy of the data. In a dynamic e-commerce environment, prices and stock levels can change rapidly. Without frequent updates, the information provided to users would quickly become stale and unreliable, undermining the core value proposition of the system. By scheduling scraping tasks via cron jobs, we guarantee that the database is populated with the most current product information.

Delta Checks: After scraping, the system compares the newly extracted price and stock data with the last recorded values in the database. Only if a significant change (e.g., a price drop beyond a threshold, or a change in stock status) is detected, is the database updated and a notification potentially triggered. This prevents unnecessary database writes and notification dispatches.

Error Recovery: The cron job system is integrated with robust error handling. If a scraping job fails (e.g., due to a website change or temporary network issue), the system can be configured to retry the job after a delay or log the error for manual intervention, ensuring data integrity is maintained even in the face of transient issues.
This automated, systematic approach to data refreshing is what transforms the system from a static price checker into a dynamic, real-time price tracking and notification service.

2.4.2 Vercel (Deployment Platform)

2.4.2.1 Serverless Functions and Frontend Hosting

Vercel is a cloud platform optimized for frontend frameworks, making it an ideal choice for deploying the Next.js-based Smart E-Commerce Price Tracking application. Its core strength lies in providing seamless hosting for static sites and serverless functions.

Frontend Hosting: Vercel automatically detects Next.js projects and deploys them with zero configuration. It optimizes assets, handles caching, and serves the static and server-rendered parts of the Next.js application globally through its CDN (Content Delivery Network). This ensures extremely fast load times for users regardless of their geographical location, contributing significantly to a smooth user experience.

Serverless Functions (API Routes): For the Next.js API routes (which handle lightweight backend logic or act as proxies to our main Node.js backend), Vercel deploys them as serverless functions. These functions are stateless, scale automatically based on demand, and only incur costs when executed. This model is highly efficient for handling sporadic requests and ensures that our API endpoints are always available and performant without managing dedicated servers.

Continuous Deployment: Vercel integrates directly with Git repositories (e.g., GitHub). Every push to the main branch automatically triggers a new deployment, and pull requests generate preview deployments. This continuous deployment (CD) workflow accelerates the development cycle, allowing for rapid iteration and testing of new features and bug fixes.
While our main Node.js/Express.js backend might reside on a separate server or a different serverless platform (like AWS Lambda or Google Cloud Functions) for more complex, long-running processes like heavy scraping, Vercel provides an excellent environment for the Next.js frontend and its integrated API routes, ensuring a highly performant and scalable user-facing application.

2.4.2.2 Continuous Deployment Workflow

Vercel's continuous deployment (CD) workflow is a cornerstone of the Smart E-Commerce Price Tracking project's development efficiency. This workflow automates the process of building, testing, and deploying code changes, significantly accelerating the release cycle and reducing manual errors.

Git Integration: The project's codebase is hosted on a Git repository (e.g., GitHub). Vercel is connected to this repository.

Automatic Builds on Push: Whenever a developer pushes new code to the main branch (or any configured branch), Vercel automatically detects the change. It then pulls the latest code, installs dependencies, and builds the Next.js application.

Instant Rollbacks: Vercel maintains a history of all deployments, making it easy to instantly roll back to a previous stable version if any issues are discovered in a new deployment.
This seamless and automated CD pipeline allows the development team to focus on writing code rather than managing complex deployment processes. It ensures that new features and bug fixes are delivered to users rapidly and reliably, which is crucial for maintaining a competitive edge in the fast-paced e-commerce environment.

2.5 Authentication and Security

2.5.1 JWT (JSON Web Tokens)

2.5.1.1 Token-Based Authentication Flow

JSON Web Tokens (JWTs) provide a concise, URL-safe means of representing claims to be transferred between two parties. In the Smart E-Commerce Price Tracking system, JWTs are central to our token-based authentication flow, offering a stateless and scalable method for user authentication.

User Login/Registration: When a user successfully logs in (via email/password or Google) or registers, the backend server verifies their credentials.

Token Generation: Upon successful verification, the server generates a JWT. This token contains a header (specifying the token type and signing algorithm), a payload (containing claims such as the user's ID (sub), email, and expiration time (exp)), and a signature (created by signing the header and payload with a secret key).

```
// Example JWT payload
{
  "sub": "user_id_123",
  "email": "user@example.com",
  "iat": 1678886400, // Issued At
  "exp": 1678890000  // Expiration Time
}
```

Token Issuance: The generated JWT is sent back to the client (frontend) in the response.

Client-Side Storage: The client stores this JWT, typically in localStorage or sessionStorage, or as an HTTP-only cookie for enhanced security.

Subsequent Requests: For every subsequent API request to protected routes (e.g., adding a product, fetching tracked items), the client includes the JWT in the Authorization header, usually as a Bearer token (Authorization: Bearer <JWT>).

Server-Side Verification: The backend server receives the request, extracts the JWT from the header, and verifies its authenticity and integrity using the secret key. It also checks the token's expiration time. If the token is valid, the server can trust the claims within the payload (e.g., the user's ID) and proceed with the request. If invalid, the request is rejected with an unauthorized error.
This flow eliminates the need for the server to store session information, making the application more scalable and resilient.

2.5.1.2 Security Considerations for Tokens

While JWTs offer significant advantages for authentication, several security considerations are paramount to prevent vulnerabilities:

Secret Key Management: The secret key used to sign the JWTs must be kept absolutely confidential and never exposed in client-side code. It should be a strong, randomly generated string and stored securely (e.g., in environment variables). Compromise of this key would allow attackers to forge valid tokens.

Token Expiration: JWTs should always have a reasonable expiration time (exp claim). Short-lived tokens reduce the window of opportunity for an attacker if a token is intercepted. For longer sessions, a refresh token mechanism can be implemented, where a long-lived refresh token is used to obtain new, short-lived access tokens.

Storage on Client-Side: Storing JWTs in localStorage or sessionStorage is susceptible to Cross-Site Scripting (XSS) attacks. If an attacker injects malicious JavaScript, they can steal the token. Using HTTP-only cookies for storing JWTs is a more secure alternative, as JavaScript cannot access them. However, this introduces CSRF (Cross-Site Request Forgery) vulnerability, which must be mitigated with CSRF tokens.

HTTPS Enforcement: All communication between the client and server must occur over HTTPS to prevent man-in-the-middle attacks from intercepting tokens.

No Sensitive Data in Payload: The JWT payload is only base64 encoded, not encrypted. Therefore, no highly sensitive information (e.g., passwords, credit card numbers) should ever be stored directly in the JWT payload. Only non-sensitive user identifiers or roles should be included.

Revocation: JWTs are inherently stateless, meaning they cannot be easily "revoked" before their expiration unless a blacklist mechanism is implemented on the server-side. For critical applications, a blacklist of revoked tokens might be necessary (e.g., for logout or password change events).

By meticulously addressing these security considerations, the JWT-based authentication system provides a robust and secure foundation for the Smart E-Commerce Price Tracking application.

2.5.2 Other Security Measures

2.5.2.1 Input Validation and Sanitization

Beyond authentication, robust input validation and sanitization are critical security measures implemented throughout the Smart E-Commerce Price Tracking system to protect against common web vulnerabilities.

Input Validation: This involves checking user-supplied data against predefined rules and constraints before it is processed or stored. For instance, when a user submits a product URL, the system validates that it is a legitimate URL format and, ideally, that it belongs to a supported e-commerce domain (e.g., Amazon.com). Similarly, email addresses are validated for correct format, and passwords are checked for complexity requirements (minimum length, special characters, etc.). Validation occurs on both the client-side (for immediate user feedback) and, crucially, on the server-side (as client-side validation can be bypassed). Libraries like Joi or express-validator in Node.js are used for server-side validation.

Input Sanitization: This process cleans or filters user input to remove or neutralize potentially malicious characters or code. For example, if a user inputs a product title that contains HTML tags or script tags, sanitization ensures these are either removed or properly escaped before being displayed or stored. This prevents Cross-Site Scripting (XSS) attacks, where an attacker injects malicious scripts into the application that are then executed by other users' browsers. Libraries like DOMPurify (for frontend) or sanitize-html (for backend) are employed for this purpose. For database interactions, using parameterized queries or ORMs like Prisma helps prevent SQL injection (or NoSQL injection for MongoDB) by properly escaping input values automatically.
By rigorously validating and sanitizing all incoming user input, the system significantly reduces its attack surface, safeguarding against data corruption, unauthorized access, and malicious code injection.

Chapter 3 PROJECT/WORK DETAILS

3.1. Project/Work Details

3.1.1 System Architecture Design

3.1.1.1 High-Level Architecture Diagram

3.1.1.1.1 Component Breakdown (Frontend, Backend, Database, Scraper, Notifier)

The Smart E-Commerce Price Tracking system is designed with a modular, distributed architecture to ensure scalability, maintainability, and high performance. At its core, the system comprises five main logical components, each with distinct responsibilities:

Frontend (User Interface):

Technology: Next.js 15, React, Tailwind CSS.

Responsibility: Provides the interactive web interface for users to register, log in, add product URLs for tracking, view current and historical price data, manage their tracked items, and configure notification preferences. It also includes the user-facing elements of the admin dashboard.

Deployment: Hosted on Vercel, leveraging SSR/SSG and serverless functions for API routes.

Backend (API Server):

Technology: Node.js, Express.js, Prisma.

Responsibility: Serves as the central API gateway for the entire application. It handles user authentication (JWT generation/validation), manages user and product data (CRUD operations), processes requests from the frontend, and orchestrates interactions with the database, scraping engine, and notification service.

Deployment: Can be deployed as a standalone Node.js server or as serverless functions (e.g., on Vercel or a dedicated cloud provider).

Database:

Technology: MongoDB.

Responsibility: Stores all persistent data, including user profiles, tracked product details (title, URL, image, current price, availability), historical price data (timestamped records), and notification logs. Its flexible schema accommodates diverse product attributes.

Deployment: Hosted on a MongoDB Atlas cluster or a self-managed instance.

Web Scraping Engine:

Technology: Node.js, Puppeteer, Cheerio, Bright Data Web Unlocker.

Responsibility: Dedicated service responsible for visiting e-commerce product pages, rendering dynamic content, bypassing anti-bot measures (via Bright Data), extracting relevant data (price, stock, title, image), and sending this data to the Backend for processing and storage.

3.1.1.2 Data Flow Overview

The data flow within the Smart E-Commerce Price Tracking system is designed to be efficient and reactive, ensuring timely updates and notifications.

User Interaction (Frontend to Backend):

A user interacts with the Frontend (Next.js app) to register, log in, or add a new product URL for tracking.

The Frontend sends API requests (e.g., POST /api/users/register, POST /api/products) to the Backend (Express.js API).

For authenticated requests, the Frontend includes the user's JWT in the Authorization header.

Authentication and Data Persistence (Backend to Database):

The Backend receives the request. For login/registration, it interacts with the Database (MongoDB) to verify credentials or create new user records.

For adding a new product, the Backend validates the URL and stores the initial product details and user association in the Database.

Automated Scraping (Cron Job to Scraping Engine to Backend to Database):

A Cron Job scheduler periodically triggers the Web Scraping Engine.

The Web Scraping Engine (Puppeteer, Cheerio, Bright Data) fetches a list of products to scrape from the Database (or receives a list from the Backend).

It then navigates to the respective e-commerce URLs, extracts the latest price, stock, and other details.

The scraped data is sent back to the Backend via an internal API call or direct function invocation.

The Backend processes this new data, compares it with existing records in the Database.

If a change (price drop, stock change) is detected, the Backend updates the product's record in the Database (including adding to priceHistory).
Notification Trigger (Backend to Notification Service):
Upon detecting a significant change in product data (e.g., price drop, stock availability change) during the scraping process, the Backend triggers the Notification Service.
The Notification Service retrieves the relevant user's notification preferences and product details from the Database.
It then composes and sends an email notification to the user.

A record of the sent notification is logged in the notifications collection in the Database.

Data Retrieval for Display (Frontend to Backend to Database):

When a user views their dashboard or a specific product page on the Frontend, it makes API requests to the Backend (e.g., GET /api/products/tracked).

The Backend retrieves the necessary product and historical price data from the Database.

This data is then sent back to the Frontend for display, including rendering price history charts.

This cyclical data flow ensures that the system is always up-to-date, responsive to user actions, and proactive in delivering critical information through automated notifications.

3.1.2 Frontend Architecture

3.1.2.1 Component Structure and State Management

The frontend of the Smart E-Commerce Price Tracking application, built with Next.js and React, adheres to a modular and component-based architecture to ensure maintainability, scalability, and reusability.

Atomic Design Principles: We largely follow Atomic Design principles, organizing components into:

Atoms: Basic HTML elements or styled components (e.g., Button, Input, Text).

Molecules: Groups of atoms functioning together (e.g., ProductCard, NotificationAlert).

Organisms: Combinations of molecules and atoms forming distinct sections of an interface (e.g., ProductDashboard, AuthForm).

Templates: Page-level structures that arrange organisms (e.g., DashboardLayout, AuthLayout).

Pages: Actual pages in the application, consuming templates and populating them with data (e.g., pages/index.js, pages/products/[id].js).

Folder Structure: The src directory is organized logically:

src/app/ (Next.js App Router structure)

src/components/: Contains reusable UI components categorized by their atomic level or functional area (e.g., components/ui, components/auth, components/products).

src/hooks/: Custom React hooks for reusable logic (e.g., useAuth, useProductTracking).

src/lib/: Utility functions, constants, API client configurations.

src/styles/: Global styles and Tailwind CSS configuration.

State Management:

Local Component State: For simple UI interactions (e.g., form input values, loading states), useState hook is used.

Global Application State (React Context API or Zustand): For data that needs to be shared across many components without prop drilling (e.g., user authentication status, global loading indicators, notification messages), the React Context API is employed. A AuthContext provides user authentication status and functions (login, logout). Alternatively, a lightweight state management library like Zustand could be used for more complex global state, offering a more performant and scalable solution for managing application-wide data.

Data Fetching State: Libraries like React Query (or SWR) are considered for managing asynchronous data fetching, caching, and synchronization with the server, which is crucial for displaying up-to-date product information and price history. This handles loading, error, and success states for API calls automatically.
This structured approach ensures a clean separation of concerns, making the codebase highly modular, testable, and easy to scale as new features are added.

3.1.2.2 API Integration Strategy

The frontend's API integration strategy is designed for efficiency, reliability, and maintainability, ensuring seamless communication with the Node.js/Express.js backend.

Centralized API Client: A dedicated API client module (src/lib/api.js or src/services/api.js) is created using axios or the native fetch API. This centralizes all API request configurations, including base URLs, headers (e.g., Content-Type, Authorization), and error handling.

Authentication Header Management: The API client automatically intercepts requests to inject the JWT (retrieved from localStorage or an HTTP-only cookie) into the Authorization: Bearer <token> header for all authenticated routes. This ensures that protected resources are accessed securely and transparently to the component logic.

Error Handling and Interceptors: Global error handling is implemented within the API client using axios interceptors. This allows us to catch common API errors (e.g., 401 Unauthorized, 404 Not Found, 500 Server Error) and perform consistent actions, such as redirecting to the login page on a 401, displaying a generic error message, or logging the error.

Custom Hooks for Data Fetching: For components that require data from the backend, custom React hooks (e.g., useProducts, useProductDetails) are created. These hooks encapsulate the API call logic, manage loading and error states, and provide the fetched data to the components. This abstracts away the data fetching complexity from the UI components, promoting reusability and cleaner code.

```
// Example: useProducts.js
import { useState, useEffect } from 'react';
import api from '../lib/api'; // Centralized API client

const useProducts = () => {
  const [products, setProducts] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchProducts = async () => {
      try {
        setLoading(true);
        const response = await api.get('/products/tracked');
        setProducts(response.data);
      } catch (err) {
        setError(err);
      } finally {
        setLoading(false);
      }
    };
    fetchProducts();
  }, []); // Dependency array ensures it runs once on mount

  return { products, loading, error };
};
```

Server-Side Data Fetching (Next.js): For initial page loads, getServerSideProps or getStaticProps (with revalidate) are used in Next.js pages to fetch data directly on the server. This improves performance and SEO by pre-rendering content before it reaches the client.
This comprehensive API integration strategy ensures that the frontend is robust, efficient, and decoupled from the backend implementation details, allowing for independent development and deployment.

3.1.3 Backend Architecture (APIs, Services)

3.1.3.1 API Endpoints Design (RESTful Principles)

The backend of the Smart E-Commerce Price Tracking system is designed around RESTful principles, providing a clear, stateless, and resource-oriented API for the frontend and internal services. Each API endpoint is carefully crafted to represent a resource and allows standard HTTP methods (GET, POST, PUT, DELETE) to perform operations on that resource.

Key API Endpoints:

User Management (/api/users):

POST /api/users/register: Register a new user.

POST /api/users/login: Authenticate a user and issue a JWT.

GET /api/users/profile: Retrieve the authenticated user's profile. (Requires JWT)

PUT /api/users/profile: Update the authenticated user's profile or notification settings. (Requires JWT)

Product Tracking (/api/products):

POST /api/products: Add a new product URL for tracking. (Requires JWT)

Request Body: { "url": "https://www.amazon.com/dp/B0..." }

Response: { "message": "Product added for tracking", "productId": "..." }

GET /api/products/tracked: Retrieve all products tracked by the authenticated user. (Requires JWT)

GET /api/products/:id: Retrieve details and historical price data for a specific tracked product. (Requires JWT)

DELETE /api/products/:id: Stop tracking a specific product. (Requires JWT)

Admin Dashboard (/api/admin - Admin-only access):

GET /api/admin/users: Retrieve a list of all registered users. (Requires Admin JWT)

PUT /api/admin/users/:id: Update a user's status or details. (Requires Admin JWT)

GET /api/admin/products: Retrieve a list of all products being tracked by the system (across all users). (Requires Admin JWT)

GET /api/admin/scraping-status: Get current status and logs of scraping jobs. (Requires Admin JWT)

POST /api/admin/scraping-trigger: Manually trigger a scraping job. (Requires Admin JWT)

Internal Scraping Callback (/api/internal/scrape-callback - Internal use only):

POST /api/internal/scrape-callback: Endpoint for the Web Scraping Engine to send scraped data back to the Backend. This endpoint would be secured by an API key or internal network access.

Request Body: { "url": "...", "price": "...", "isAvailable": "...", "title": "...", "imageUrl": "...", "timestamp": "..." }

This design ensures a clear, predictable, and scalable API surface, making it easy for the frontend to consume data and for future services to integrate with the system.

3.1.3.2 Service Layer for Business Logic
To maintain a clean and maintainable codebase, the backend architecture incorporates a distinct service layer that encapsulates the core business logic, separating it from the API route handlers. This separation of concerns is crucial for several reasons:

Modularity and Reusability: Business logic (e.g., user registration, product tracking, price change detection, notification triggering) is contained within dedicated service files (e.g., userService.js, productService.js, scrapingService.js, notificationService.js). These services can be reused across different API endpoints or even by internal cron jobs without duplicating code.

Testability: By isolating business logic in services, it becomes much easier to write unit tests for specific functionalities without needing to spin up the entire Express.js server or mock complex request/response objects. This improves the quality and reliability of the codebase.

Maintainability: Changes to business rules or database interactions only require modifications within the relevant service file, minimizing the risk of introducing bugs in other parts of the application. This makes the system easier to understand, debug, and extend.

Scalability: A clear service layer can facilitate the transition to a microservices architecture in the future, where each service could potentially become an independent microservice.

Example Structure:

```
/src
  /api
    /routes
      users.js    // Express routes for user-related APIs
      products.js // Express routes for product-related APIs
      admin.js    // Express routes for admin-related APIs
    index.js      // Combines all routes
  /services
    userService.js      // Handles user registration, login, profile updates
    productService.js      // Handles adding/deleting products, retrieving user-tracked products
    scrapingService.js  // Orchestrates scraping, processes scraped data, detects changes
    notificationService.js // Handles sending emails, managing notification logs
  /db
    prisma.js          // Prisma client instance
    schema.prisma       // Prisma schema definition
  /utils
    jwt.js             // JWT token generation/verification
    errorHandler.js     // Centralized error handling
```

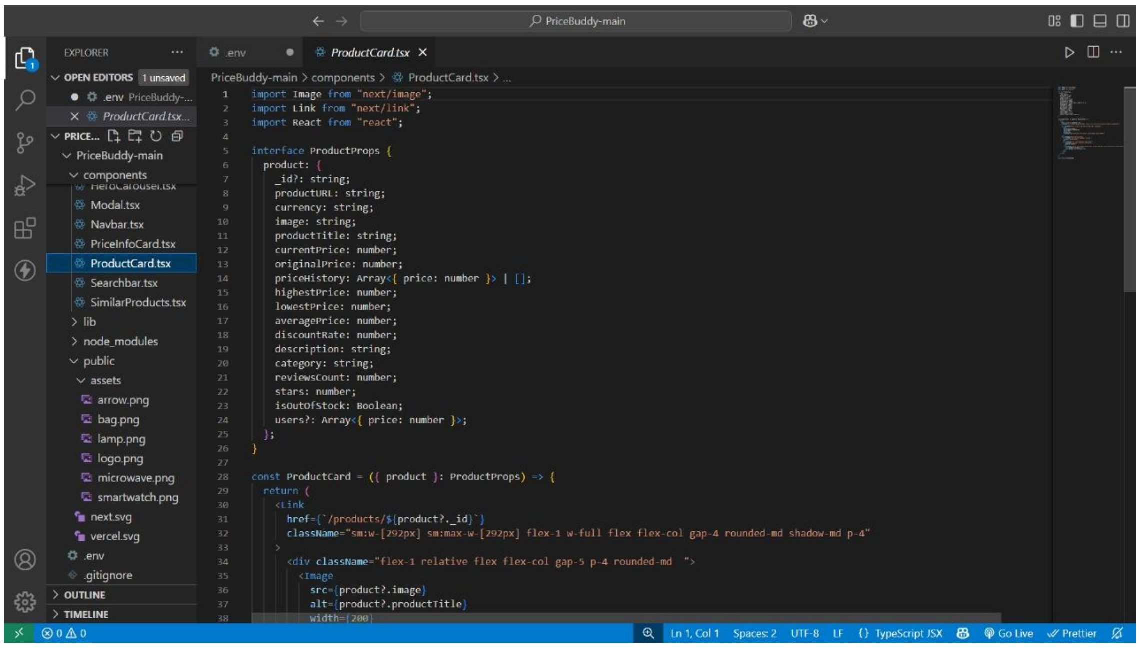
Example Flow:
An API route handler (e.g., products.js) would receive a request, perform basic validation, and then delegate the core logic to a service function (e.g., productService.addProduct). The service function would then interact with the database (via Prisma) and potentially other services (e.g., scrapingService to initiate an initial scrape for the new product). This layered approach ensures that the API routes remain thin, primarily focusing on request parsing and response formatting, while the heavy lifting of business logic is handled by the dedicated service layer.

3.1.4 Database Schema Design (MongoDB Collections)

The MongoDB schema design is optimized for the flexible and evolving nature of e-commerce product data, ensuring efficient storage and retrieval of price tracking information. We primarily utilize three main collections: users, products, and notifications.

3.1.4.1 User Collection Schema

The users collection stores all user-specific information, including authentication credentials, personal preferences, and a reference to the products they are tracking. This design allows for quick retrieval of user profiles and their associated tracked items.



```tsx
import Image from "next/image";
import Link from "next/link";
import React from "react";

interface ProductProps {
  product: {
    _id?: string;
    productURL: string;
    currency: string;
    image: string;
    productTitle: string;
    currentPrice: number;
    originalPrice: number;
    priceHistory: Array<{ price: number }> | [];
    highestPrice: number;
    lowestPrice: number;
    averagePrice: number;
    discountRate: number;
    description: string;
    category: string;
    reviewsCount: number;
    stars: number;
    isOutOfStock: Boolean;
    users?: Array<{ price: number }>;
  };
}

const ProductCard = ({ product }: ProductProps) => {
  return (
    <Link
      href={`/products/${product?._id}`}
      className="sm:w-[292px] sm:max-w-[292px] flex-1 w-full flex flex-col gap-4 rounded-md shadow-md p-4"
    >
      <div className="flex-1 relative flex flex-col gap-5 p-4 rounded-md  ">
        <Image
          src={product?.image}
          alt={product?.productTitle}
          width={200}
```

Figure 3 User Collection Schema

3.2 Implementation Details - User Features

3.2.1 Secure Authentication Flow

3.2.1.1 Email/Password Registration and Login Implementation

The implementation of email/password registration and login is a cornerstone of the Smart E-Commerce Price Tracking system, providing a robust and secure entry point for users.

Registration:

Frontend Form: The registration page presents a form for users to input their email and desired password (and password confirmation).

Client-Side Validation: Basic validation (e.g., email format, password strength) is performed on the client-side for immediate feedback.

API Call: Upon submission, the frontend sends a POST request to /api/users/register with the email and password.

Server-Side Validation: The backend receives the request and performs comprehensive validation:

Checks for existing email (ensuring uniqueness).

Validates email format and password strength (e.g., minimum length, presence of special characters, numbers).

Password Hashing: If validation passes, the user's password is hashed using bcrypt with a strong salt. This ensures that plain-text passwords are never stored in the database.

User Creation: A new user document is created in the users MongoDB collection with the hashed password.

JWT Generation & Response: A JWT is generated for the newly registered user, containing their _id and email. This token is sent back to the frontend.

Login:

Frontend Form: The login page presents a form for email and password input.

API Call: The frontend sends a POST request to /api/users/login with the user's credentials.

Server-Side Verification: The backend retrieves the user record based on the provided email.

Password Comparison: It then uses bcrypt.compare() to compare the provided password (after hashing it with the stored salt) against the stored hashed password.

JWT Generation & Response: If the passwords match, a new JWT is generated and sent back to the frontend.

Client-Side Token Handling: Upon receiving the JWT, the frontend stores it securely (e.g., in localStorage or an HTTP-only cookie) and redirects the user to their dashboard. For subsequent authenticated requests, this token is included in the Authorization header. This entire process ensures that user credentials are handled securely from input to storage and subsequent authentication.

3.2.1.2 Google Authentication Integration Steps

Integrating Google Authentication provides a convenient and widely adopted alternative for users to sign up and log in, enhancing the user experience by reducing friction. The implementation typically involves OAuth 2.0 flow:

Google Developer Console Setup:

Create a new project in the Google Cloud Console.

Enable the Google People API.

Configure OAuth 2.0 client IDs for a "Web application," specifying authorized JavaScript origins (e.g., http://localhost:3000, https://your-app.vercel.app) and authorized redirect URIs. This provides a client ID and client secret.

Frontend (Next.js):

Use a library like @react-oauth/google or implement a custom Google Sign-In button.

When the user clicks "Sign in with Google," a pop-up or redirect to Google's authentication server occurs.

Upon successful authentication by Google, the frontend receives an id_token (a JWT issued by Google) and potentially an access_token.

The id_token is then sent to our backend.

Backend (Node.js/Express.js):

An API endpoint (e.g., POST /api/users/google-auth) receives the id_token from the frontend.

The backend uses Google's official client libraries (e.g., google-auth-library) to verify the id_token. This verification ensures the token is valid, hasn't expired, and was issued by Google for our specific client ID.

From the verified id_token payload, the backend extracts user information such as email, Google ID, and name.

User Management:

Checks if a user with that Google ID already exists in our users collection. If so, it's a login.

If not, it checks if a user with that email already exists (e.g., they previously registered with email/password). If so, it links the Google ID to their existing account.

If neither exists, a new user account is created in our database with the Google ID and email.

JWT Generation & Response: Our backend then generates its own internal JWT for this user (containing our _id and email) and sends it back to the frontend.
This seamless integration allows users to leverage their existing Google accounts, streamlining the onboarding process while maintaining our internal authentication and authorization mechanisms.

3.2.1.3 JWT Token Generation and Validation

The secure generation and validation of JWTs are critical for maintaining the integrity and authorization within the Smart E-Commerce Price Tracking system.

JWT Generation (Backend):
When a user successfully logs in or registers, the backend generates a JWT using the jsonwebtoken library.

Payload Creation: A payload object is constructed, containing non-sensitive user-specific claims. Essential claims include:

sub (subject): The unique ID of the user from our MongoDB _id.

email: The user's email address.

iat (issued at): Timestamp of when the token was issued.

exp (expiration time): Timestamp when the token expires (e.g., 1 hour from issuance).

Optionally, role: If an admin role exists, it would be included here.

Signing: The payload is then signed using a strong, secret key stored securely in environment variables. The signing algorithm (e.g., HS256) is also specified.

```
const jwt = require('jsonwebtoken');
const SECRET_KEY = process.env.JWT_SECRET_KEY; // Loaded from environment
variables

const generateToken = (userId, email, role) => {
  const payload = {
    sub: userId,
    email: email,
    role: role // 'user' or 'admin'
  };
  // Sign the token with the secret key and set expiration
  return jwt.sign(payload, SECRET_KEY, { expiresIn: '1h' });
};
```

The generated token is then sent to the client.

JWT Validation (Backend Middleware):
For every protected API route, a JWT validation middleware is executed before the actual route handler.

Token Extraction: The middleware extracts the JWT from the Authorization header (Bearer <token>).

Verification: The jsonwebtoken.verify() method is used to verify the token's signature using the same secret key. This step also automatically checks the exp claim, throwing an error if the token has expired.

Payload Decryption: If verification is successful, the method returns the decoded payload.

User Context: The decoded user information (e.g., userId, email, role) is attached to the req.user object, making it accessible to subsequent route handlers.

Authorization Check: For routes requiring specific roles (e.g., admin routes), an additional check is performed on req.user.role.

```
const verifyToken = (req, res, next) => {
  const authHeader = req.headers.authorization;
  if (!authHeader || !authHeader.startsWith('Bearer ')) {
    return res.status(401).json({ message: 'No token provided' });
  }
  const token = authHeader.split(' ')[1];
  try {
    const decoded = jwt.verify(token, SECRET_KEY);
    req.user = decoded; // Attach user info to request
    next(); // Proceed to the next middleware/route handler
  } catch (error) {
    if (error.name === 'TokenExpiredError') {
      return res.status(401).json({ message: 'Token expired' });
    }
    return res.status(403).json({ message: 'Invalid token' });
  }
};

// Example usage in an Express route:
// router.get('/profile', verifyToken, (req, res) => { /* req.user available here */ });
```

This robust generation and validation process ensures that only legitimate and authorized users can access protected resources, forming the foundation of the system's security.

3.2.2 Product Tracking Functionality

3.2.2.1 User Interface for URL Input

The user interface for adding products to track is designed to be intuitive and straightforward, minimizing friction for the user.

Dedicated Input Section: On the user's main dashboard or a dedicated "Add Product" page, a prominent input field is provided, clearly labeled "Enter Product URL" or "Track a New Product."

Clear Call to Action: A distinct button, such as "Track Product," "Add to Watchlist," or "Start Tracking," accompanies the input field.

Validation Feedback: As the user types or after submission, immediate feedback is provided:

Valid URL Format: If the input is not a valid URL, a real-time error message appears (e.g., "Please enter a valid URL").

Supported Merchant: If the URL is valid but from an unsupported e-commerce site, a message like "Currently, only Amazon.com URLs are supported" is displayed.

Loading Indicator: After submission, a loading spinner or message ("Fetching product details...") appears, indicating that the system is processing the request.

Success/Error Messages: Upon successful addition, a confirmation message appears (e.g., "Product added successfully!"). If there's an error (e.g., product not found, scraping failed), an informative error message is shown.

Example Input: Placeholder text or a small example URL (e.g., https://www.amazon.com/dp/B0...) is provided to guide the user on the expected input format.

Responsive Design: The input section is designed to be fully responsive, ensuring it looks and functions well on both desktop and mobile devices, with adequate padding and clear typography.
This user-centric design ensures that users can easily and confidently add products to their tracking list, making the initial interaction with the system seamless.

3.2.2.2 Backend Logic for Adding/Managing Tracked Products
The backend logic for adding and managing tracked products is a core component of the system, orchestrating the interaction between user requests, the database, and the scraping engine.

Adding a New Product:

API Endpoint: A POST request to /api/products is received from the frontend, containing the product url in the request body.

Authentication & Authorization: The request is first passed through JWT authentication middleware to ensure the user is logged in.

URL Validation: The backend rigorously validates the provided URL:

Checks for valid URL format.

Extracts the domain to verify it's from a supported e-commerce merchant (e.g., amazon.com).

Extracts unique product identifiers (e.g., ASIN for Amazon) to prevent duplicate tracking of the same product.

Duplicate Check: The system queries the products collection to check if this specific product URL is already being tracked by any user. If it exists, it simply links the current user to the existing product, avoiding redundant scraping for the same item.

Initial Scraping: If the product is new to the system:

The backend triggers an initial, immediate scrape for this product using the Web Scraping Engine. This is a crucial step to get the current price and details right away.

The scraping engine returns the title, imageUrl, currentPrice, isAvailable, and merchant details.

Error handling is critical here: if the initial scrape fails (e.g., product not found, website issue), an appropriate error is returned to the user.

Database Insertion/Update:

If the product is new, a new document is created in the products collection, including the scraped details and the initial price point in the priceHistory array.

The _id of this new product document is then added to the trackedProductIds array of the requesting user's document in the users collection.

If the product already existed, the user's trackedProductIds array is simply updated.

Response: A success response is sent back to the frontend, confirming the product has been added for tracking.

Managing Tracked Products (User-Specific):

Retrieving User's Tracked Products: A GET request to /api/products/tracked retrieves all products associated with the authenticated user. The backend queries the users collection for the user's trackedProductIds and then fetches the corresponding product documents from the products collection, including their priceHistory.

Stopping Tracking: A DELETE request to /api/products/:id allows a user to remove a product from their tracking list. The backend removes the productId from the user's trackedProductIds array. If no other user is tracking this product, the product document itself might be soft-deleted or marked for archival to clean up the database.
This backend logic ensures efficient, secure, and accurate management of user-tracked products, forming the core data management for the application.

3.2.2.3 Displaying Tracked Product Information

The presentation of tracked product information to the user is designed for clarity, conciseness, and actionable insights. This is primarily handled on the user's dashboard.

Product Cards/Listings: Each tracked product is displayed as a distinct card or row in a list format. Key information presented includes:

Product Image: A thumbnail of the product image, fetched from the scraped imageUrl.

Product Title: The scraped title of the product.

Current Price: The most recently scraped price, prominently displayed.

Availability Status: A clear indicator (e.g., "In Stock," "Out of Stock," or a colored icon) showing the product's current availability.

Last Updated: A timestamp indicating when the price/stock was last refreshed, assuring the user of data freshness.

Price Change Indicator: A visual cue (e.g., a green arrow pointing down for a price drop, a red arrow pointing up for an increase, or a neutral icon) showing the price trend since the last check or since the user started tracking.

Detailed Product View: Clicking on a product card navigates the user to a dedicated product detail page (/products/[id]). This page provides:

All the information from the product card, but in a larger, more detailed format.

Interactive Price History Chart: A full-width chart (using a charting library like Recharts) displaying the product's price fluctuations over time, allowing users to zoom, pan, and see specific price points and dates. This is crucial for understanding trends.

3.2.3.3 Email Notification Service Integration (e.g., Nodemailer, SendGrid)
The email notification service is the final crucial step in delivering timely alerts to users. It integrates with a third-party email sending solution to ensure reliable delivery.

Choice of Service:

Nodemailer: For simpler setups or if using a transactional email service (like Mailgun, SendGrid, or AWS SES) via SMTP. Nodemailer is a module for Node.js applications to allow easy email sending. It's highly flexible and supports various transport methods.

SendGrid API: For more robust, scalable, and feature-rich email delivery. SendGrid offers a dedicated API, analytics, template management, and high deliverability rates, which are crucial for a production-grade notification system. Its Node.js client library simplifies integration.

Implementation (notificationService.js):

API Call/Function Invocation: The notificationService is called by the backend (e.g., scrapingService) when a price drop or stock change is detected. It receives parameters such as userId, productId, type, and relevant price/availability details.

Retrieve User & Product Data: It fetches the user's email and notification preferences from the users collection and the product's title, image, and URL from the products collection.

Email Content Generation:

Templates: HTML email templates are used to create professional and consistent notification emails. These templates are populated dynamically with product-specific data (e.g., product title, old price, new price, direct link to product page, image).

Personalization: The email is personalized with the user's name (if available) and relevant details.

Sending Email:

Using Nodemailer: Configures an SMTP transporter with credentials and then uses transporter.sendMail() to send the email.

Using SendGrid API: Utilizes the SendGrid Node.js client (@sendgrid/mail) to make an API call to SendGrid's servers, passing the recipient, subject, and HTML content.

Error Handling & Logging: The service includes robust error handling for email sending failures (e.g., invalid recipient email, API issues). Failed attempts are logged in the notifications collection with status: 'failed' and errorDetails for debugging. Successful sends are logged with status: 'sent'.

```
// Simplified SendGrid example
const sgMail = require('@sendgrid/mail');
sgMail.setApiKey(process.env.SENDGRID_API_KEY);

const sendPriceDropNotification = async (userEmail, productTitle, oldPrice, newPrice,
productUrl) => {
  const msg = {
    to: userEmail,
    from: 'noreply@your-app.com', // Verified sender email
    subject: `Price Drop Alert: ${productTitle}`,
    html: `
      <p>The price of <strong>${productTitle}</strong> has dropped!</p>
      <p>Old Price: $${oldPrice}</p>
      <p>New Price: <strong>$${newPrice}</strong></p>
      <p><a href="${productUrl}">View Product on Amazon</a></p>
      <p>Happy shopping!</p>
    `,
  };
  try {
    await sgMail.send(msg);
    console.log(`Email sent to ${userEmail} for ${productTitle}`);
    // Log success in notifications collection
  } catch (error) {
    console.error(`Error sending email to ${userEmail}:`, error);
    // Log failure in notifications collection
  }
};
```

This integration ensures that users receive timely, professional, and reliable notifications, which is a key value proposition of the price tracking system.

each user; instead, it creates a many-to-many relationship where multiple users can track the same product, and a single product document exists in the products collection.

User-Specific Settings: Future enhancements could include adding more granular user preferences within the users collection, such as a targetPrice for a specific product, or custom notes for a wishlisted item. This would require embedding a sub-document within the trackedProductIds array or creating a separate userProductPreferences collection.

```
// Example of enhanced trackedProductIds with user-specific settings
"trackedProducts": [
  {
    "productId": ObjectId("65c8f1e7b9f3d2a1b4c5e6d8"),
    "addedDate": ISODate("2024-05-01T10:00:00Z"),
    "targetPrice": 450.00, // User-defined target price for this product
    "notes": "Birthday gift for Mom",
    "isFavorite": true // Boolean to mark as a 'favorite'
  }
]
```

This database integration ensures that each user's tracked products are uniquely identified, easily managed, and retrieved efficiently, forming the basis of their personalized price tracking experience.

Out of Stock Indicator: Clearly visible "Out of Stock" labels or different styling for unavailable items.

Confirmation Modals: When a user attempts to remove a product, a small confirmation modal (Dialog component from Shadcn/UI) pops up, asking "Are you sure you want to stop tracking this product?" to prevent accidental removals.

Empty State: If a user has no products tracked, a friendly message is displayed, guiding them on how to add their first product.
This comprehensive UI ensures that users have complete control over their tracked products, making the management of their wishlist efficient and user-friendly.

3.3 Implementation Details - Web Scraping & Automation

3.3.1 Web Scraping Engine Design

3.3.1.1 Initializing Bright Data Web Unlocker

The Bright Data Web Unlocker is the first line of defense against anti-scraping measures and is initialized at the very beginning of our scraping process. Its primary role is to act as an intelligent proxy that handles all the complexities of web page rendering, CAPTCHA solving, IP rotation, and browser fingerprinting, allowing our core scraping logic to receive clean, fully rendered HTML.

Configuration: The Web Unlocker is configured with specific parameters, including the target domain (e.g., amazon.com), desired output format (HTML), and any specific browser emulation settings. This is typically done by setting up a proxy endpoint provided by Bright Data.

Authentication: Our scraping requests are authenticated with Bright Data using a unique username and password provided upon account creation. This ensures that only authorized requests are processed through their network.

Integration with HTTP Clients (Fetch/Axios): For pages that are largely static or where initial HTML contains most data, we configure our HTTP requests (using fetch or axios in Node.js) to route through the Bright Data proxy. This involves setting the proxy agent in axios or using a custom agent with fetch that points to the Web Unlocker's endpoint.

Integration with Puppeteer: For highly dynamic e-commerce sites that rely heavily on JavaScript rendering, Puppeteer is launched with proxy arguments pointing to the Bright Data Web Unlocker. This ensures that the headless browser's traffic is routed through Bright Data's network, benefiting from its anti-blocking capabilities. The Puppeteer instance also authenticates with Bright Data.

```
const puppeteer = require('puppeteer');
`http://${process.env.BRIGHT_DATA_USERNAME}:${process.env.BRIGHT_DATA
_PASSWORD}@${process.env.BRIGHT_DATA_HOST}:${process.env.BRIGHT_D
ATA_PORT}`;

async function launchBrowserWithBrightData() {
  const browser = await puppeteer.launch({
    args: [`--proxy-server=${BRIGHT_DATA_PROXY}`],
    headless: true, // Use 'new' for new headless mode in Puppeteer v21+
    // ... other browser options
  });
  return browser;
}
```

By initializing and routing all scraping traffic through the Bright Data Web Unlocker, we significantly enhance the reliability and success rate of our data extraction, allowing the system to consistently gather up-to-date information from challenging e-commerce environments.

3.3.1.2 Puppeteer Scripting for Page Navigation and Rendering

Puppeteer scripting is essential for interacting with dynamic e-commerce websites that load content asynchronously or require user interactions to reveal full product details.

Browser and Page Initialization: A headless Chromium browser instance is launched using Puppeteer. A new page (tab) is then opened for each product to be scraped.

Navigation and Waiting: The page.goto(url, { waitUntil: 'domcontentloaded' }) method is used to navigate to the target product URL. Crucially, waitUntil: 'domcontentloaded' or waitUntil: 'networkidle0' (for more complex SPAs) ensures that the page's initial HTML and all necessary network requests have completed, allowing JavaScript to render the content.

Dynamic Content Loading: Many e-commerce sites load prices, stock status, or even product descriptions via AJAX requests after the initial page load. Puppeteer handles this by:

page.waitForSelector(selector): Waits until a specific CSS selector (e.g., the price element) appears in the DOM, indicating that the content has been rendered.

page.waitForFunction(function): Executes a JavaScript function within the browser context and waits until it returns a truthy value. This is useful for waiting for complex conditions, like a specific data attribute to be present or a loading spinner to disappear.

3.3.3 Data Storage and Historical Tracking

3.3.3.1 Storing Core Product Details (Name, Image, Current Price, URL)

The efficient and structured storage of core product details is fundamental to the Smart E-Commerce Price Tracking system. These details are stored in the products collection within MongoDB, designed to be easily accessible and updateable.

Unique Identification: Each product document is uniquely identified by its _id (MongoDB's ObjectId) and its url (which has a unique index to prevent duplicate entries).

Essential Fields: The following core details are captured and stored for each product:
url (String): The full URL of the product page on the e-commerce website. This is the primary key for tracking.

title (String): The full, human-readable title of the product as scraped from the page.
imageUrl (String): The URL of the product's main image. This is crucial for displaying product cards in the UI.

currentPrice (Number): The most recently scraped numerical price of the product. This is updated with every successful scrape.
currency (String): The currency symbol or code (e.g., "USD", "INR") associated with the currentPrice.

isAvailable (Boolean): A flag indicating the current stock status (true for in stock, false for out of stock).

merchant (String): The name of the e-commerce platform (e.g., "Amazon", "Flipkart") from which the product was scraped.

lastUpdated (Date): A timestamp indicating the exact time of the last successful scrape for this product, crucial for determining data freshness and scheduling future scrapes.

createdAt (Date): Timestamp when the product was first added to the tracking system.

Dynamic Schema Advantage: MongoDB's flexible schema allows us to easily add optional fields in the future (e.g., category, brand, descriptionSnippet, ratings) without requiring schema migrations, adapting to the diverse data available on different e-commerce sites.
This structured approach ensures that all essential product information is readily available for display, analysis, and notification triggering.

3.3.3.2 Implementing Price History Logging (Timestamped Records)

One of the most valuable features of the Smart E-Commerce Price Tracking system is its ability to log and display historical price data. This is implemented by embedding a priceHistory array directly within each product document in the products collection.

Embedded Array Structure: The priceHistory array is an array of sub-documents, where each sub-document represents a single recorded price point at a specific timestamp.

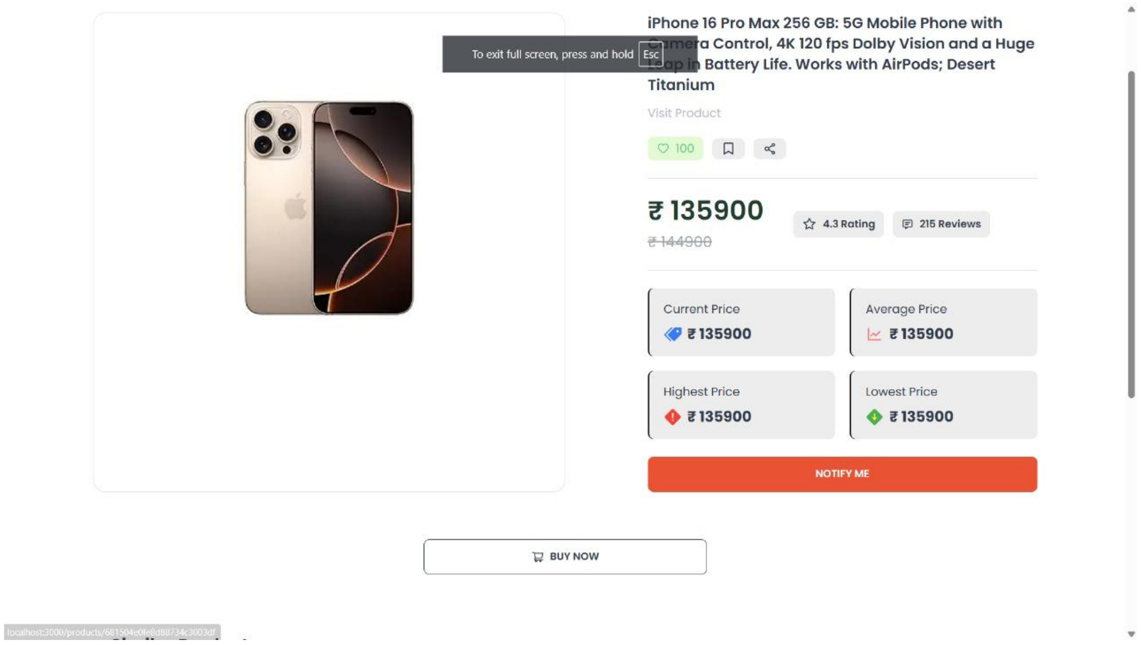

iPhone 16 Pro Max 256 GB: 5G Mobile Phone with Camera Control, 4K 120 fps Dolby Vision and a Huge Leap in Battery Life. Works with AirPods; Desert Titanium

Figure 4 Timestamped Records

Logging Mechanism:

After each successful scrape, the backend compares the newly scraped currentPrice with the currentPrice already stored in the database.

If the currentPrice has changed (or if a significant period has passed since the last log, even if the price is the same, to ensure regular data points), a new entry { price: newPrice, timestamp: new Date() } is appended to the priceHistory array.

The currentPrice field in the main product document is also updated.

Querying for History: When a user views a product's details page, the backend fetches the entire product document, including the priceHistory array. This array is then passed to the frontend charting library (e.g., Recharts) to render the interactive price trend graph.

Data Volume Management: For products tracked over very long periods, the priceHistory array could grow large. Strategies to manage this include:

Capping Array Size: Limiting the priceHistory array to a certain number of recent entries (e.g., the last 100 entries) or a specific time window (e.g., last 6 months). Older entries could be archived or summarized.

Downsampling: For very old data, storing daily or weekly averages instead of every single scrape.
This implementation ensures that users have access to a comprehensive historical view of price changes, empowering them to make more informed purchasing decisions based on trends.

3.3.3.3 Tracking Stock Status Changes
Tracking stock status changes is as vital as price tracking, especially for high-demand products that frequently go in and out of stock. This is managed directly within the products collection.

isAvailable Field: Each product document has a boolean field, isAvailable, which represents the most current stock status scraped from the e-commerce website.

true: Product is currently in stock.

false: Product is currently out of stock.

Update Mechanism:

During each scheduled scrape, the Web Scraping Engine extracts the current stock status from the product page. This might involve looking for specific text ("In Stock," "Out of Stock," "Temporarily Unavailable") or the presence/absence of an "Add to Cart" button.

The backend compares the newly scraped isAvailable status with the isAvailable status stored in the database.

If a change is detected (e.g., oldIsAvailable: false and newIsAvailable: true), the isAvailable field in the product document is updated.

```
"stockHistory": [
  { "status": false, "timestamp": ISODate("2024-05-10T10:00:00Z") }, // Went out of
stock
  { "status": true, "timestamp": ISODate("2024-05-15T14:00:00Z") }, // Back in stock
  { "status": false, "timestamp": ISODate("2024-05-20T09:00:00Z") }  // Went out of
stock again
]
```

3.4 Implementation Details - Admin Dashboard

3.4.1 User Management Module

3.4.1.1 Admin Interface for Viewing/Editing User Data

The Admin Dashboard's User Management Module provides administrators with a centralized and intuitive interface to oversee and manage all registered users within the Smart E-Commerce Price Tracking system.

User List Table: The core of this module is a sortable and filterable table displaying a comprehensive list of all users. Key columns include:

User ID (_id)

Email Address

Registration Date (createdAt)

Last Login Date (optional, derived from login logs)

Number of Tracked Products (derived by counting trackedProductIds array length)

Account Status (e.g., "Active," "Suspended")

Actions (Edit, Deactivate/Activate, Delete)

Search and Filter: A search bar allows admins to quickly find users by email or ID. Filters enable sorting by registration date, number of tracked products, or account status.

User Profile View/Edit: Clicking on a user in the table opens a detailed view or modal that allows admins to:

View full user details (email, registration date, notification settings).

Edit user's notification preferences (e.g., toggle price drop emails).

Change account status (e.g., deactivate an account if it's found to be spamming or misusing the service).

Reset password (by sending a password reset link, not directly changing it).

Permissions and Roles: The interface ensures that only users with an "admin" role can access this module, and specific actions (like deleting users) might require higher-level admin permissions.

Audit Trails (Optional): For critical actions (e.g., deactivating an account), an audit trail could log which admin performed the action and when, enhancing accountability.
The interface is built using Next.js components and Tailwind CSS, ensuring responsiveness and a consistent design. Data is fetched from the backend via secure API endpoints (e.g., GET /api/admin/users, PUT /api/admin/users/:id), which are protected by admin-specific JWT validation.

3.4.2 Product Management Module

3.4.2.1 Admin View of All Tracked Products

The Product Management Module within the Admin Dashboard provides administrators with a comprehensive overview of all products currently being tracked across the entire system, regardless of which user is tracking them. This centralized view is crucial for system monitoring, debugging, and understanding overall market trends.

Master Product List Table: A sortable and filterable table displays every unique product document from the products collection. Key columns include:

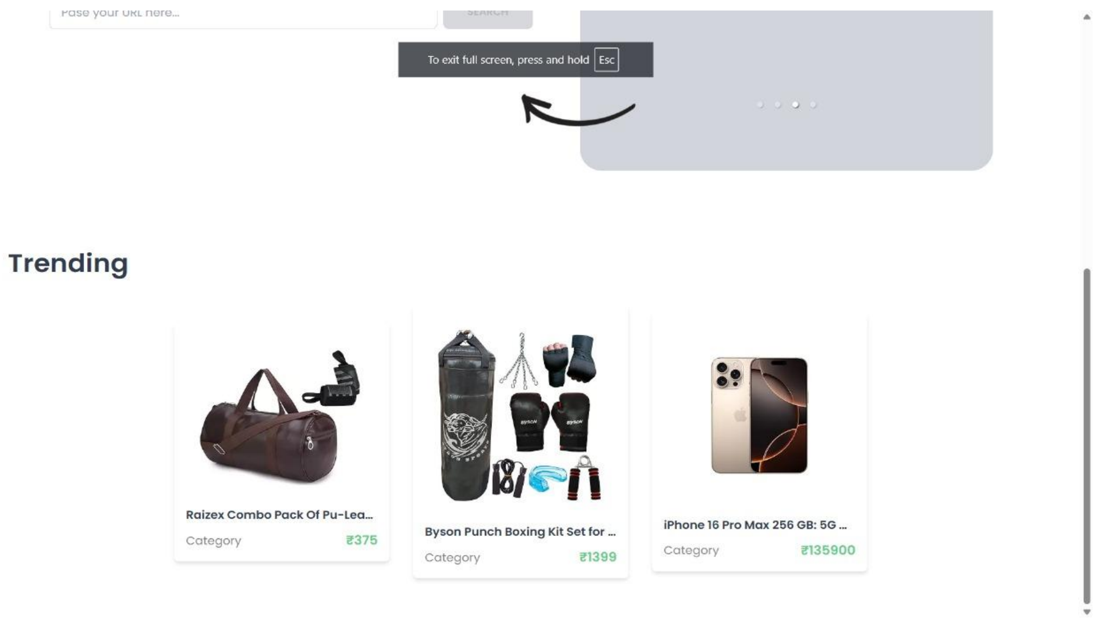

Pase your URL here...
SEARCH
To exit full screen, press and hold Esc
Trending
Raizex Combo Pack Of Pu-Lea...
Category
₹375
Byson Punch Boxing Kit Set for ...
Category
₹1399
iPhone 16 Pro Max 256 GB: 5G ...
Category
₹135900

Figure 5 Admin View of All Tracked Products

Chapter 4 RESULTS/OUTCOME

4.1 System Implementation and Functional Verification

This subsection thoroughly details each implemented feature, verifying its functionality against the project's requirements. It showcases the successful development of the system's core components.

4.1.1 User Authentication and Authorization System

This segment outlines the robust security mechanisms implemented for user access and management.

4.1.1.1 User Registration Process:

Description: Detail the user sign-up flow, emphasizing its simplicity and security. Explain how new users create accounts using their email addresses, including password complexity requirements and email verification steps.

Verification: Provide screenshots of the registration form and a successful registration confirmation. Discuss any measures taken to prevent bot registrations.

4.1.1.2 Secure Login Mechanisms:

Description: Elaborate on the user login interface and the underlying security protocols. Discuss the use of JWT-based authentication for secure session management. Explain how user credentials are handled securely (e.g., password hashing).

Verification: Include screenshots of the login page and a successful user dashboard view post-login. Demonstrate robustness against common login vulnerabilities.

4.1.1.3 Google OAuth Integration:

Description: Explain the seamless integration with Google for single sign-on (SSO). Discuss the benefits of this approach for user convenience and reduced credential management.

Verification: Show the Google login prompt and the resulting user experience.

4.1.1.4 Role-Based Access Control (RBAC):

Description: Explain how different user roles (e.g., standard user, administrator) have distinct access levels and functionalities within the application. This is crucial for the admin dashboard features.

Verification: Illustrate with screenshots showing the differences in UI/features between a standard user and an admin account.

4.1.2 Product Tracking and Notification Core

This section focuses on the central functionality of the system: enabling users to monitor products and receive alerts.

4.1.2.1 Product Link Submission and Validation:

Description: Detail the mechanism by which users input product URLs (e.g., Amazon links). Describe the backend validation process to ensure valid e-commerce links and initial data parsing.

Verification: Show the user interface for adding a product. Provide examples of successful and unsuccessful link submissions, highlighting error handling.

4.1.2.2 Real-time Price Monitoring Implementation:

Description: Elaborate on the web scraping engine's configuration and operation using Bright Data Web Unlocker, Cheerio, and Puppeteer. Discuss how the system fetches and parses product data, including price, title, image, and availability. Explain the logic for identifying price changes.

Verification: Present screenshots of the system's internal logs showing successful data retrieval. Compare scraped data with actual product pages to confirm accuracy.

4.1.2.3 Automated Price Drop Alert System:

Description: Detail the logic and implementation of automated email notifications for price reductions. Discuss the email templating and delivery service used.

Verification: Include examples of received email alerts. Discuss the trigger conditions (e.g., any price drop, drop below a certain threshold).

4.1.2.4 Stock Availability Notification System:

Description: Explain how the system detects changes in product stock status and triggers notifications when a product is back in stock.

Verification: Show a log of stock status changes and corresponding notification triggers.

4.1.2.5 User Wishlist and Favorites Management:

Description: Describe the functionality allowing users to save and organize products for tracking, acting as a personal watchlist.

Verification: Provide screenshots of the user's wishlist interface, demonstrating product addition, removal, and viewing.

4.1.3 Web Scraping and Automation Efficiency

This subsection delves into the technical performance and reliability of the data acquisition process.

4.1.3.1 Web Scraping Engine Performance:

Description: Detail the configuration and performance tuning of Bright Data Web Unlocker, Cheerio, and Puppeteer. Discuss strategies employed to handle dynamic content, CAPTCHAs, and anti-scraping measures.

Verification: Present metrics such as average scraping time per product, success rate of data extraction, and resource utilization (CPU, memory) during scraping operations.

4.1.3.2 Automated Cron Job Reliability:

Description: Explain the setup and scheduling of periodic cron jobs that ensure up-to-date pricing information and stock status. Discuss the chosen frequency of updates and the rationale behind it.

Verification: Show logs demonstrating the consistent execution of cron jobs over time. Report on the uptime and reliability of the cron job service.

4.1.3.3 Database Management and Data Integrity:

Description: Detail the schema design and implementation of MongoDB for storing historical price trends and tracking product availability. Discuss how data consistency and integrity are maintained.

Verification: Present sample database entries. Demonstrate data retrieval speeds for historical price data. Discuss backup and recovery strategies.

4.1.4 Admin Dashboard Functionality

This section highlights the tools available to administrators for managing the system and its users.

4.1.4.1 Product Management Interface:

Description: Detail the features allowing admins to view, edit, add, or remove tracked products directly. Discuss filtering and searching capabilities for large datasets.

Verification: Provide screenshots of the product management dashboard. Demonstrate product editing flows.

4.1.4.2 User Management and Preferences:

Description: Explain how administrators can manage registered users, including viewing their tracking preferences, account status, and potentially suspending or deleting accounts.

Verification: Show the user management interface and capabilities like user search and status toggling.

4.1.4.3 Scraping Control Panel:

Description: Detail the adjustable scraping frequency and failure monitoring features. Explain how admins can intervene in scraping processes, e.g., restarting failed jobs or adjusting global scraping parameters.

Verification: Provide screenshots of the control panel. Show examples of monitoring logs for scraping errors.

4.1.4.4 Data Analytics and Reporting:

Description: Discuss the insights provided into price trends, user engagement, and notification effectiveness. Explain the types of reports available (e.g., most tracked products, popular price drops, notification success rates).

Verification: Include screenshots of charts and graphs from the analytics dashboard. Present key metrics derived from the collected data.

4.2 System Performance and Technical Metrics

This subsection provides quantitative evidence of the system's operational efficiency and responsiveness.

4.2.1 Frontend Performance and User Experience

4.2.1.1 UI Responsiveness and Loading Times:

Description: Discuss the responsiveness of the user interface across different screen sizes and devices, achieved using Next.js 15 and Tailwind CSS. Present metrics on page load times for key sections of the application.

Verification: Utilize tools like Google Lighthouse reports or GTmetrix to show performance scores (e.g., First Contentful Paint, Largest Contentful Paint).

4.2.1.2 Navigational Fluidity:

Description: Evaluate the smoothness and speed of user navigation between different sections of the application (e.g., dashboard, wishlist, settings).

Verification: User session recordings or click-through rate analysis if available.

4.2.2 Backend API Performance

4.2.2.1 API Endpoint Latency:

Description: Provide performance metrics for critical API endpoints developed with Node.js, Express.js, and Prisma. Focus on response times for user authentication, product addition, and data retrieval.

Verification: Use tools like Postman, Apache JMeter, or other load testing frameworks to measure average response times under various load conditions.

4.2.2.2 Scalability under Load:

Description: Discuss the backend's ability to handle an increasing number of concurrent users and requests. Explain how the chosen technologies contribute to scalability.

Verification: Present results from stress tests demonstrating the system's behavior near its capacity limits, showing how it maintains performance or gracefully degrades.

4.2.3 Database Query Performance

4.2.3.1 Data Retrieval Speed:

Description: Analyze the performance of MongoDB queries, especially for fetching historical price data and large sets of tracked products.

Verification: Show query execution times for common database operations. Discuss indexing strategies employed to optimize performance.

4.2.3.2 Database Load and Resource Utilization:

Description: Monitor MongoDB's resource consumption (CPU, memory, disk I/O) under typical and peak loads.

Verification: Present graphs showing resource utilization over time.

4.2.4 Notification System Throughput and Reliability

4.2.4.1 Email Delivery Latency:

Description: Report on the average time taken from a price drop/stock change detection to the delivery of the corresponding email notification.

Verification: Logs from your email service provider.

4.2.4.2 Notification Success Rate:

Description: Present the percentage of successfully delivered notifications versus attempts, highlighting any bounces or failures.

Verification: Data extracted from your notification system's logs.

4.3 Security Posture and Deployment Success

This subsection validates the security measures and the successful deployment of the application.

4.3.1 Security Measures Efficacy

4.3.1.1 JWT-based Authentication Validation:

Description: Detail the implementation of JWT for secure user access and session management. Discuss its benefits (statelessness, scalability) and how it protects against common web vulnerabilities.

Verification: Explain penetration testing results related to authentication.

4.3.1.2 Data Encryption and Protection:

Description: Explain how sensitive user data (e.g., passwords) is encrypted both in transit and at rest. Discuss measures taken to protect against data breaches.

Verification: Outline the encryption algorithms and protocols used (e.g., HTTPS, bcrypt for passwords).

4.3.1.3 Input Validation and Sanitization:

Description: Describe how user inputs (especially product URLs) are validated and sanitized to prevent injection attacks (e.g., SQL injection, XSS).

Verification: Discuss results from security scans or manual testing for these vulnerabilities.

4.3.2 Cloud Deployment Outcomes

4.3.2.1 Vercel Deployment Process:

Description: Detail the steps taken for deploying the Next.js application on Vercel. Discuss the advantages of using a platform like Vercel for continuous deployment and scalability.

Verification: Provide screenshots of the deployment pipeline and successful deployment status.

4.3.2.2 Application Uptime and Availability:

Description: Report on the continuous uptime and availability of the deployed application.

Verification: Present uptime monitoring reports from services like UptimeRobot.

4.3.2.3 CDN and Global Reach Performance:

Description: Discuss how Vercel's CDN enhances content delivery and response times for users geographically distributed.

Verification: Provide network latency tests from various global locations.

4.4 User Feedback and Project Impact

This final subsection quantifies the positive outcomes and user reception of the system.

4.4.1 User Experience and Satisfaction

4.4.1.1 Usability Test Results:

Description: Summarize findings from any user testing sessions. Detail observed user behaviors, pain points, and positive feedback regarding the interface and functionality.

Verification: Present a summary of user survey responses or qualitative feedback collected during testing.

Chapter 5 CONCLUSION & FUTURE WORK

5.1 Project Conclusion

This subsection synthesizes the entire project, highlighting its core contributions and successful outcomes.

5.1.1 Summary of Key Achievements

5.1.1.1 Successful Problem Resolution:

Description: Reiterate how the "Smart E-Commerce Price Tracking" system effectively addresses the challenges faced by consumers in tracking price fluctuations and stock availability.

Highlights: Emphasize the automated, real-time nature of the solution.

5.1.1.2 Robust System Development:

Description: Summarize the successful development and integration of a full-stack web application, highlighting the effective use of Next.js 15, Bright Data Web Unlocker, MongoDB, Node.js, and Express.js.

Highlights: Mention the user-friendly interface and secure authentication mechanisms.

5.1.1.3 Tangible Benefits Delivered:

Description: Briefly recap the main benefits achieved, such as improved consumer decision-making, competitive advantage for sellers, and significant time/cost savings.

Highlights: Stress the automation aspect that reduces manual effort.

5.1.2 Challenges Encountered and Solutions

5.1.2.1 Web Scraping Robustness:

Description: Discuss the inherent challenges of web scraping, such as dealing with dynamic website structures, CAPTCHAs, and anti-bot measures.

Solutions: Explain how Bright Data Web Unlocker, combined with Puppeteer and Cheerio, was instrumental in overcoming these hurdles, ensuring reliable data extraction.

5.1.2.2 Real-time Notification Scalability:

Description: Address the complexities of implementing a scalable and timely notification system that can handle a growing number of users and products.

Solutions: Detail how cron jobs and a robust email delivery service were configured to ensure efficiency and reliability.

5.1.2.3 Data Storage and Management:

Description: Outline challenges in storing and managing historical price data efficiently in MongoDB.

Solutions: Discuss schema design, indexing strategies, and database optimization techniques employed.

5.2 Future Work

This subsection outlines a clear roadmap for the project's continued development, envisioning future enhancements and strategic expansions.

5.2.1 Immediate Enhancements and Feature Extensions

5.2.1.1 Multi-Platform Support:

Description: Prioritize expanding the web scraping capabilities to include other major e-commerce platforms beyond Amazon, such as Flipkart, eBay, and Myntra. This would significantly broaden the system's utility.

Implementation Considerations: Discuss the need for adaptable scraping modules and potential challenges unique to each new platform.

5.2.1.2 Advanced Notification Customization:

Description: Allow users more granular control over notification settings. This could include choosing notification frequency (e.g., daily digest, instant alerts), setting specific price thresholds for alerts (e.g., alert only if price drops by 10%), or opting for different notification channels (e.g., SMS, push notifications via a mobile app).

Technical Implications: Requires integration with SMS gateways or push notification services.

5.2.1.3 Enhanced Admin Analytics and Reporting:

Description: Develop more sophisticated data visualization tools and customizable reporting features within the admin dashboard. This could include predictive trend analysis for overall market prices, user demographic insights, and more detailed notification effectiveness reports.

Benefits: Provides deeper operational insights and supports strategic decision-making.

5.2.2 Long-Term Vision and AI/ML Integration

5.2.2.1 AI-based Price Prediction:

Description: Integrate machine learning models to forecast future price trends for tracked products. This would move the system beyond reactive alerts to proactive intelligence.

Methodology: Discuss potential ML algorithms (e.g., time series analysis, regression models) and the need for large historical datasets from MongoDB.

Benefits: Enables users to anticipate optimal purchase times, enhancing their decision-making even further.

5.2.2.2 Dedicated Mobile Application Development:

Description: Create native or cross-platform mobile applications (iOS and Android) to provide users with a more convenient, always-on experience.

Technologies: Suggest frameworks like React Native or Flutter for efficient cross-platform development.

Features: Push notifications, quick access to wishlists, and mobile-optimized interfaces.

5.2.2.3 Voice Assistant Integration:

Description: Add support for popular voice assistants like Amazon Alexa and Google Assistant, enabling users to query product prices or add items to their tracking list using voice commands.

Technical Requirements: Integration with respective voice assistant APIs and natural language processing (NLP) capabilities.

5.2.3 Scalability and System Architecture Enhancements

5.2.3.1 Microservices Architecture Migration:

Description: As the system grows, consider transitioning from a monolithic architecture to a microservices-based approach. This would improve modularity, fault isolation, and independent scalability of components (e.g., separate services for scraping, notifications, user management).

Benefits: Enhanced development agility, resilience, and horizontal scalability.

5.2.3.2 Advanced Caching Mechanisms:

Description: Implement more aggressive caching strategies at various layers (e.g., CDN, application-level, database caching) to further reduce load times and improve responsiveness for frequently accessed data.

5.2.3.3 Enhanced Error Handling and Logging:

Description: Develop a more sophisticated centralized logging and error reporting system to quickly identify and diagnose issues in production.

References

[1] Gandomi, A., & Haider, M. (2015). Beyond the hype: Big data concepts, methods, and analytics. *International Journal of Information Management*, *35*(2), 137–144. https://doi.org/10.1016/j.ijinfomgt.2014.10.007

[2] Shmueli, G., Bruce, P. C., Gedeck, P., & Patel, N. R. (2020). *Data Mining for Business Analytics: Concepts, Techniques, and Applications in Python* (3rd ed.). Wiley.

[3] Aggarwal, C. C. (2016). *Recommender Systems: The Textbook*. Springer. https://doi.org/10.1007/978-3-319-29659-3

[4] Zhang, Y., & Pennacchiotti, M. (2013). Predicting purchase behaviors from social media. *Proceedings of the 22nd International Conference on World Wide Web*, 1521–1532. https://doi.org/10.1145/2488388.2488501

[5] Tan, P. N., Steinbach, M., Karpatne, A., & Kumar, V. (2019). *Introduction to Data Mining* (2nd ed.). Pearson.

[6] Armstrong, G., Kotler, P., Harker, M., & Brennan, R. (2020). *Marketing: An Introduction* (14th ed.). Pearson Education.

[7] Laudon, K. C., & Traver, C. G. (2023). *E-commerce 2023: Business, Technology, Society* (18th ed.). Pearson.

[8] Montgomery, A. L., & Srinivasan, K. (2003). Learning about customers without asking. *Harvard Business Review*, *81*(5), 58–65.

[9] Anderson, C. (2006). *The Long Tail: Why the Future of Business Is Selling Less of More*. Hyperion.

[10] Choudhury, V., & Karahanna, E. (2008). The relative advantage of electronic channels: A multidimensional view. *MIS Quarterly*, *32*(1), 179–200.

[11] Bakos, Y. (1997). Reducing buyer search costs: Implications for electronic marketplaces. *Management Science*, *43*(12), 1676–1692.

[12] Brynjolfsson, E., & Smith, M. D. (2000). Frictionless commerce? A comparison of Internet and conventional retailers. *Management Science*, *46*(4), 563–585.

[13] Varian, H. R. (2014). *Intermediate Microeconomics: A Modern Approach* (9th ed.). W.W. Norton.

[14] Chen, Y., Narasimhan, C., & Zhang, Z. J. (2001). Individual marketing with imperfect targetability. *Marketing Science*, *20*(1), 23–41.

[15] Sun, B., Li, S., & Zhou, C. (2006). Adaptive learning and proactive targeting in dynamic markets. *Management Science*, *52*(5), 713–730.

[16] Kannan, P. K., & Li, H. (2017). Digital marketing: A framework, review, and research agenda. *International Journal of Research in Marketing*, *34*(1), 22–45.

[17] Koutroumpis, P. (2009). The economic impact of broadband on growth: A simultaneous approach. *Telecommunications Policy*, *33*(9), 471–485.

[18] Li, H., Kuo, C., & Russell, M. G. (1999). The impact of perceived channel utilities, shopping orientations, and demographics on the consumer's online buying behavior. *Journal of Computer-Mediated Communication*, *5*(2). https://doi.org/10.1111/j.1083-6101.1999.tb00336.x

[19] Hosanagar, K., & Tan, Y. (2016). Platform competition under asymmetric information. *Management Science*, *62*(4), 1052–1067.

[20] Ghosh, R., & McAfee, P. (2012). Incentivizing high-quality user-generated content. *Proceedings of the 22nd International Conference on World Wide Web,* 137–147.

[21] Kietzmann, J. H., Hermkens, K., McCarthy, I. P., & Silvestre, B. S. (2011). Social media? Get serious! Understanding the functional building blocks of social media. *Business Horizons, 54*(3), 241–251.

[22] Rust, R. T., & Chung, T. S. (2006). Marketing models of service and relationships. *Marketing Science, 25*(6), 560–580.

[23] Parise, S., Guinan, P. J., & Weinberg, B. D. (2008). The secrets of marketing in a Web 2.0 world. *Wall Street Journal*, December.

[24] Pindyck, R. S., & Rubinfeld, D. L. (2018). *Microeconomics* (9th ed.). Pearson.

[25] Li, X., & Karahanna, E. (2015). Online recommendation systems in electronic commerce. *MIS Quarterly, 39*(2), 409–428.

[26] Kumar, V., & Reinartz, W. (2016). *Creating Enduring Customer Value*. Journal of Marketing, *80*(6), 36–68.

[27] Bhargava, H. K., & Sundaresan, S. (2004). Pricing mechanisms for online retailing. *Journal of Revenue and Pricing Management, 3*(4), 371–379.

[28] Park, Y. J., & Gretzel, U. (2007). Success factors for destination marketing websites. *Journal of Travel Research, 46*(1), 46–63.

[29] Li, S., Sun, B., & Wilcox, R. T. (2005). Cross-selling sequentially ordered products. *Journal of Marketing Research, 42*(3), 355–362.

[30] Häubl, G., & Trifts, V. (2000). Consumer decision making in online shopping environments: The effects of interactive decision aids. *Marketing Science*, *19*(1), 4–21.

[31] Wu, G., & Rangaswamy, A. (2003). A fuzzy set model of search and consideration with an application to an online market. *Marketing Science, 22*(3), 411–434.

[32] Urban, G. L., Sultan, F., & Qualls, W. J. (2000). Placing trust at the center of your internet strategy. *Sloan Management Review, 42*(1), 39–48.

[33] Palmatier, R. W., Dant, R. P., Grewal, D., & Evans, K. R. (2006). Factors influencing the effectiveness of relationship marketing: A meta-analysis. *Journal of Marketing, 70*(4), 136–153.

[34] Dodds, W. B., Monroe, K. B., & Grewal, D. (1991). Effects of price, brand, and store information on buyers' product evaluations. *Journal of Marketing Research, 28*(3), 307–319.

[35] Kahneman, D. (2011). *Thinking, Fast and Slow*. Farrar, Straus and Giroux.

[36] Riggins, F. J. (1999). A framework for identifying Web-based electronic commerce opportunities. *Journal of Organizational Computing and Electronic Commerce, 9*(4), 297–310.

[37] Iyer, B., & Davenport, T. H. (2008). Reverse engineering Google's innovation machine. *Harvard Business Review, 86*(4), 58–68.

[38] Chen, Y., Pavlov, D., & Canny, J. (2009). Large-scale behavioral targeting. *Proceedings of the 15th ACM SIGKDD,* 209–218.

[39] Srinivasan, S. S., Anderson, R. E., & Ponnavolu, K. (2002). Customer loyalty in e-commerce: An exploration of its antecedents and consequences. *Journal of Retailing, 78*(1), 41–50.

[40] Kumar, V., Aksoy, L., Donkers, B., Venkatesan, R., Wiesel, T., & Tillmanns, S. (2010). Undervalued or overvalued customers: Capturing total customer engagement value. *Journal of Service Research, 13*(3), 297–310.

[41] Zeng, D. D., Chen, H., Lusch, R., & Li, S. H. (2010). Social media analytics and intelligence. *IEEE Intelligent Systems, 25*(6), 13–16.

[42] Sarwar, B., Karypis, G., Konstan, J., & Riedl, J. (2001). Item-based collaborative filtering recommendation algorithms. *Proceedings of the 10th International Conference on World Wide Web*, 285–295.

[43] Breese, J. S., Heckerman, D., & Kadie, C. (1998). Empirical analysis of predictive algorithms for collaborative filtering. *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence*, 43–52.

[44] Xu, H., Teo, H. H., Tan, B. C., & Agarwal, R. (2010). The role of push-pull technology in privacy calculus: The case of location-based services. *Journal of Management Information Systems, 26*(3), 135–174.

[45] Belanger, F., Hiller, J. S., & Smith, W. J. (2002). Trustworthiness in electronic commerce: The role of privacy, security, and site attributes. *The Journal of Strategic Information Systems, 11*(3-4), 245–270.

[46] Liu, H., Keeling, D. I., & Papamichail, K. N. (2020). Shaping business intelligence through big data analytics: A case study in retail. *Decision Support Systems, 138*, 113383.

[47] Lee, K., Park, J., & Han, I. (2011). The different effects of online consumer reviews on consumers' purchase intentions depending on trust in online shopping malls. *Internet Research, 21*(2), 187–206.

[48] Kotler, P., Keller, K. L., & Chernev, A. (2022). *Marketing Management* (16th ed.). Pearson.

[49] Ghose, A., & Ipeirotis, P. G. (2011). Estimating the helpfulness and economic impact of product reviews: Mining text and reviewer characteristics. *IEEE Transactions on Knowledge and Data Engineering, 23*(10), 1498–1512.

[50] Trusov, M., Bodapati, A. V., & Bucklin, R. E. (2010). Determining influential users in Internet social networks. *Journal of Marketing Research, 47*(4), 643–658.

[51] R.E. Higgs, K.G. Bemis, I.A. Watson and J.H. Wikel, Experimental designs for selecting molecules from large chemical databases", Journal of Chemical Information and Computer Sciences, 37(5), 861-870, September 1997.

[52] N. Paivinen and T. Gronfors, Minimum spanning tree clustering of EEG signals", 6th Nordic Signal Processing Symposium (NORSIG-2004), Finland, pp. 149-152, June 9-11, 2004.

[53] Guide to Technical Report Writing: University of Sussex, Study Guides: School of Engineering & Informatics, Dr. H. Prance, http://www.sussex.ac.uk/ei/internal/forstudents/engineeringdesign/studyguides/techreportwriting, accessed 23 July 2018.