1. Introduction

1.1. Objective

The objective of this project is to design and implement a scalable, high-performance backend for a modern e-commerce marketplace, similar to Amazon. The system is built using Node.js, Express, and a MongoDB NoSQL database, focusing on efficient schema design and advanced query capabilities.

1.2. Scope

This project includes the complete design of the database schema, a strategy for efficient indexing, and the implementation of a RESTful API. The key features include managing products, users, orders, and reviews. A primary focus is a powerful search functionality, including keyword search, fuzzy search, and an advanced hybrid ranking system (implemented as a bonus) that leverages MongoDB Atlas Search.

---

2. Question 1: System Design

2.1. MongoDB Schema Design & Justification

The database is composed of four main collections: users, products, orders, and reviews. The design balances data normalization and query performance by using a mix of referenced and embedded documents.

```
const mongoose = require('mongoose');

const UserSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true, index: true },
  location: { type: String },
}, { timestamps: true });

module.exports = mongoose.model('User', UserSchema);
```

- **User Schema:** This schema is straightforward. The email field is indexed and set as unique to ensure data integrity and fast lookups during login or when referencing a user. The purchase history is *not* stored here; it is derived at query time from the orders collection to maintain scalability.

```
const mongoose = require('mongoose');

const ProductSchema = new mongoose.Schema({
  name: { type: String, required: true },
  description: { type: String },
  category: { type: String, index: true },
  price: { type: Number, required: true, index: true },
  brand: { type: String, index: true },
  rating: { type: Number, default: 0 },
  stock: { type: Number, default: 0 },
  purchaseCount: { type: Number, default: 0 },
}, { timestamps: true });


ProductSchema.index({ name: 'text', description: 'text' });

ProductSchema.index({ category: 1, price: 1 });
ProductSchema.index({ brand: 1, price: 1 });
ProductSchema.index({ purchaseCount: -1 });

module.exports = mongoose.model('Product', ProductSchema);
```

- **Product Schema:** This schema includes all product details and several indexes to optimize various search and filtering queries. It contains a purchaseCount field, which is intended to be incremented on each purchase, to provide a simple metric for popularity-based ranking.

```
• const mongoose = require('mongoose');

  const OrderItemSchema = new mongoose.Schema({
    product: { type: mongoose.Schema.Types.ObjectId, ref: 'Product',
  required: true },
    quantity: { type: Number, required: true, min: 1 },
    price: { type: Number, required: true }
  });

  const OrderSchema = new mongoose.Schema({
    user: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required:
  true, index: true },
    items: [OrderItemSchema],
    totalCost: { type: Number, required: true },
  }, { timestamps: true });

  OrderSchema.index({ createdAt: -1 });

  module.exports = mongoose.model('Order', OrderSchema);
```

Order Schema Justification (Referencing vs. Embedding):

- **Referencing:** The user field is a reference to the users collection using its ObjectId. This is a normalized approach. If a user's name or location changes, it only needs to be updated in the single users document, not in every order they have ever placed.
- **Embedding:** The items array is **embedded** directly within the Order document. This is a denormalized approach chosen for performance. An order's items are part of that single order; they will always be queried together and are not accessed individually. Embedding them avoids a costly database join and allows all order details to be fetched in a single read operation.

```
• const mongoose = require('mongoose');

  const ReviewSchema = new mongoose.Schema({
    user: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required:
```

```
true },
  product: { type: mongoose.Schema.Types.ObjectId, ref: 'Product',
required: true, index: true },
  rating: { type: Number, required: true, min: 1, max: 5 },
  text: { type: String },
}, { timestamps: true });

module.exports = mongoose.model('Review', ReviewSchema);
```

- **Review Schema:** This schema follows a fully referenced model, linking to both the user who wrote the review and the product it is for. This is highly scalable and allows for queries to easily find all reviews for a product or all reviews by a user.

2.2. Indexing Strategy

Indexes are crucial for optimizing query performance. The following indexes were designed to support the application's key read operations:

- **ProductSchema.index({ name: 'text', description: 'text' })**
    - **Justification:** This text index is the foundation of the GET /products/search endpoint. It allows MongoDB to perform efficient, stemmed keyword searches across both the product name and description fields.
- **OrderSchema.index({ user: 1 })**
    - **Justification:** This index is critical for the GET /users/:id/orders endpoint. It allows the database to instantly find all orders belonging to a specific user ID without scanning the entire collection.
- **ReviewSchema.index({ product: 1 })**
    - **Justification:** This index directly supports the GET /products/:id/reviews endpoint, enabling a fast lookup of all reviews associated with a specific product ID.
- **ProductSchema.index({ purchaseCount: -1 })**
    - **Justification:** This index is essential for the bonus hybrid search. It allows the aggregation pipeline to quickly sort products by popularity (purchaseCount) in descending order.
- **ProductSchema.index({ category: 1, price: 1 })**
    - **Justification:** This compound index is designed to optimize common user browsing patterns, such as filtering by a category and then sorting the results by price.

2.3. API Design

The core functionality of the marketplace is exposed via the following RESTful API endpoints:

Method	Endpoint	Description
GET	/products/search?query=	Searches for products by keyword using the text index.
GET	/users/:id/orders	Retrieves a list of all orders for a specific user.
GET	/products/:id/reviews	Retrieves all reviews for a specific product.
GET	/orders/:id	Retrieves the full, populated details of a single order.
GET	/products/search/hybrid?query=	(Bonus) Performs an advanced hybrid search by similarity, popularity, and price.

2.4. Aggregation Pipeline

To find the "Top 5 most frequently purchased products in the last month, grouped by category," the following aggregation pipeline was designed for the GET /reports/top-products endpoint.

```
app.get('/reports/top-products', async (req, res) => {
  try {
    const oneMonthAgo = new Date();
    oneMonthAgo.setDate(oneMonthAgo.getDate() - 30);

    const pipeline = [
      {

        $match: {
          createdAt: { $gte: oneMonthAgo }
        }
      },
      {
        $unwind: '$items'
      },
      {
        $group: {
          _id: '$items.product',
          totalPurchased: { $sum: '$items.quantity' }
        }
      },
      {

        $sort: {
          totalPurchased: -1
        }
      },
      {

        $lookup: {
          from: 'products',
          localField: '_id',
          foreignField: '_id',
          as: 'productDetails'
        }
      },
      {

        $unwind: '$productDetails'
      },
      {

        $group: {
          _id: '$productDetails.category',
          products: {

            $push: {
              productId: '$_id',
              name: '$productDetails.name',
              totalPurchased: '$totalPurchased'
            }
          }
        }
      },
      {

        $project: {
          _id: 0,
          category: '$_id',
          top5Products: {
            $slice: ['$products', 5]
          }
        }
      },
      {

        $sort: {
          category: 1
        }
      }
    ];


    const result = await Order.aggregate(pipeline);
    res.json(result);

  } catch (err) {
    console.error(err);
    res.status(500).json({ message: 'Aggregation Error', error: err.message
});
  }
});
```
Stage-by-Stage Explanation:

1. **$match**: First, this stage filters the entire orders collection to find only those documents created within the last 30 days.

2. **$unwind**: This stage deconstructs the items array from each order, creating a separate document for each item purchased.
3. **$lookup**: This stage performs a "left join" to the products collection. It uses the items.product ID to find the matching product and retrieve its category.
4. **$unwind**: A second unwind is needed to deconstruct the productDetails array created by the $lookup.
5. **$group (First)**: This is the main grouping stage. It groups the documents by both category and productId, using $sum to count the total quantity purchased for each unique product.
6. **$sort**: This stage sorts the products *within* their groups by the totalPurchased count in descending order (most popular first).
7. **$group (Second)**: This stage re-groups the sorted products, this time by category only. It uses $push to push all the product information (ID, name, total) into a new array called products.
8. **$project**: The final stage cleans up the output. It uses $slice to select only the top 5 elements from each category's products array, perfectly fulfilling the requirement.

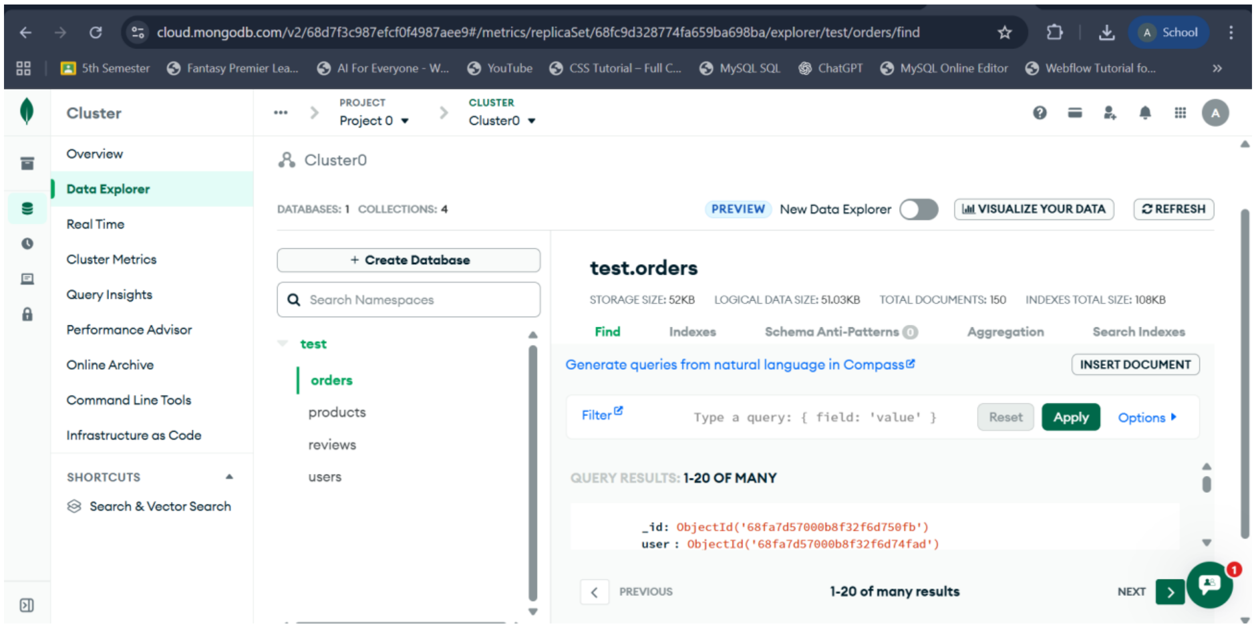
---

3. Question 2: Implementation & Evidence

This section provides the execution screenshots as evidence that the designed system was successfully implemented.

3.1. Database Seeding

The MongoDB Atlas cloud database was populated using a custom seed.js script. The following screenshot shows the ecommerce_project database with its collections (users, products, orders, reviews) filled with data.



test.orders

STORAGE SIZE: 52KB LOGICAL DATA SIZE: 51.03KB TOTAL DOCUMENTS: 150 INDEXES TOTAL SIZE: 108KB

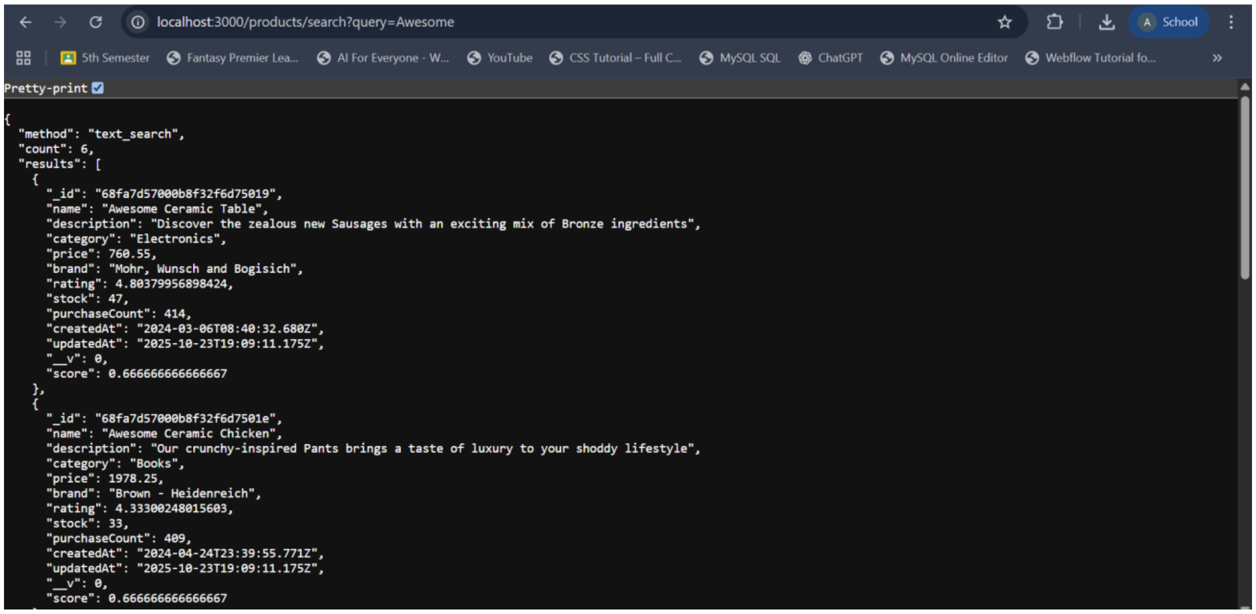
QUERY RESULTS: 1-20 OF MANY

```
_id: ObjectId('68fa7d57000b8f32f6d750fb')
user : ObjectId('68fa7d57000b8f32f6d74fad')
```

3.2. API & Search Implementation

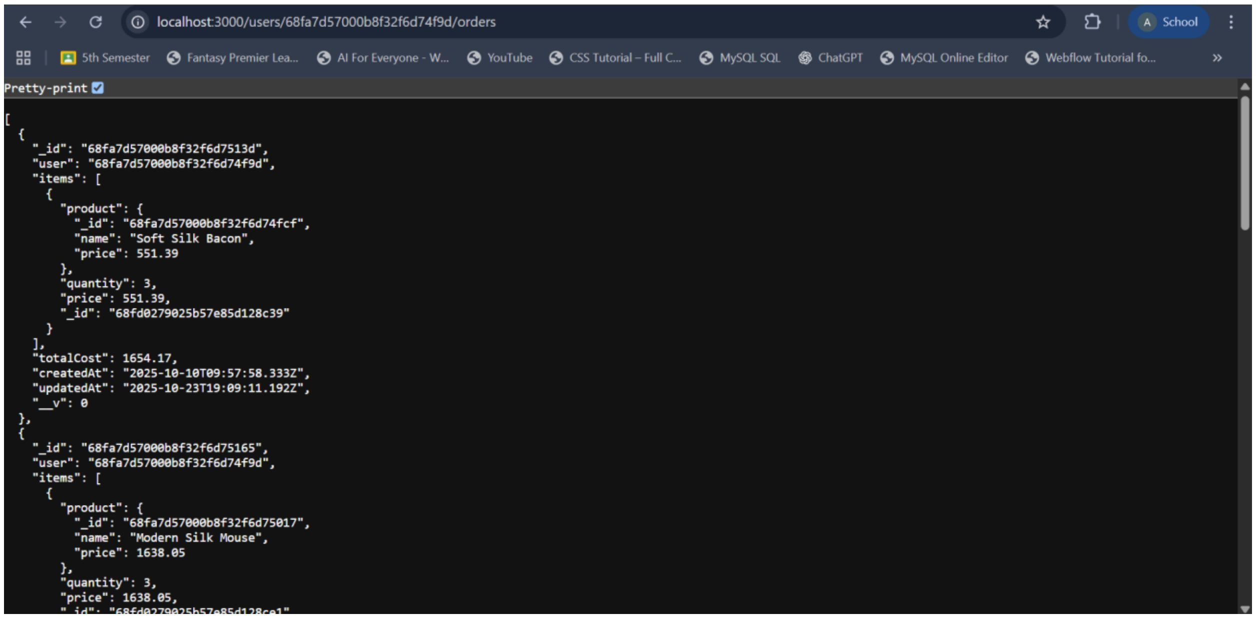
The implemented API endpoints were tested by sending live requests to the running server.

1. Standard Search: The GET /products/search endpoint successfully uses the text index to find matching products.



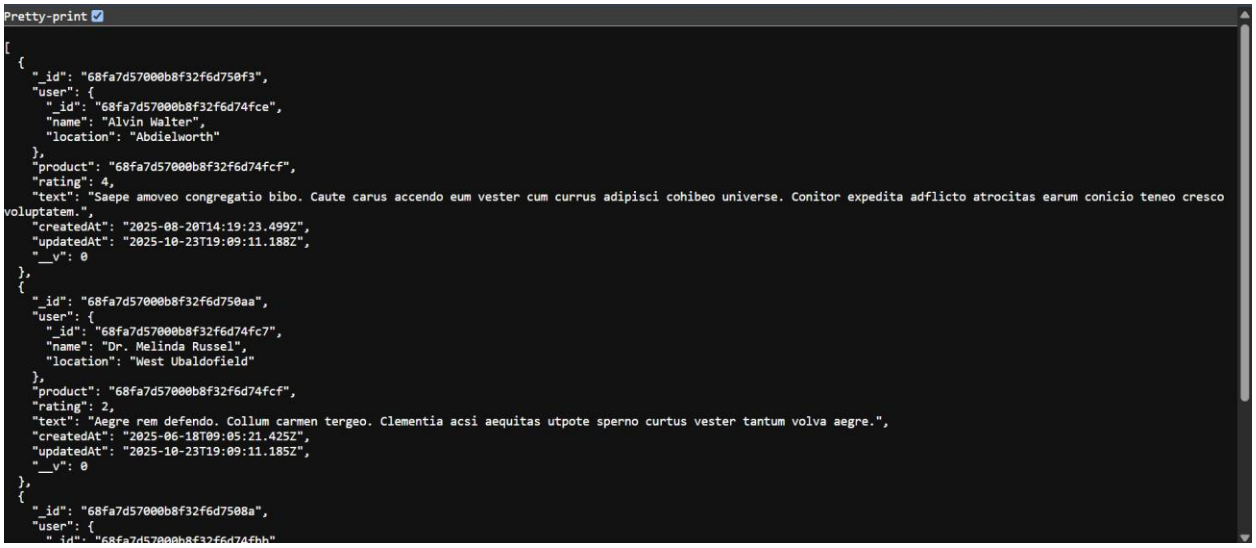
```
{
  "method": "text_search",
  "count": 6,
  "results": [
    {
      "_id": "68fa7d57000b8f32f6d75019",
      "name": "Awesome Ceramic Table",
      "description": "Discover the zealous new Sausages with an exciting mix of Bronze ingredients",
      "category": "Electronics",
      "price": 760.55,
      "brand": "Mohr, Wunsch and Bogisich",
      "rating": 4.80379956898424,
      "stock": 47,
      "purchaseCount": 414,
      "createdAt": "2024-03-06T08:40:32.680Z",
      "updatedAt": "2025-10-23T19:09:11.175Z",
      "__v": 0,
      "score": 0.666666666666667
    },
    {
      "_id": "68fa7d57000b8f32f6d7501e",
      "name": "Awesome Ceramic Chicken",
      "description": "Our crunchy-inspired Pants brings a taste of luxury to your shoddy lifestyle",
      "category": "Books",
      "price": 1978.25,
      "brand": "Brown - Heidenreich",
      "rating": 4.33300248015603,
      "stock": 33,
      "purchaseCount": 409,
      "createdAt": "2024-04-24T23:39:55.771Z",
      "updatedAt": "2025-10-23T19:09:11.175Z",
      "__v": 0,
      "score": 0.666666666666667
```

2. User Orders: The GET /users/:id/orders endpoint successfully fetches all orders for a specific user.



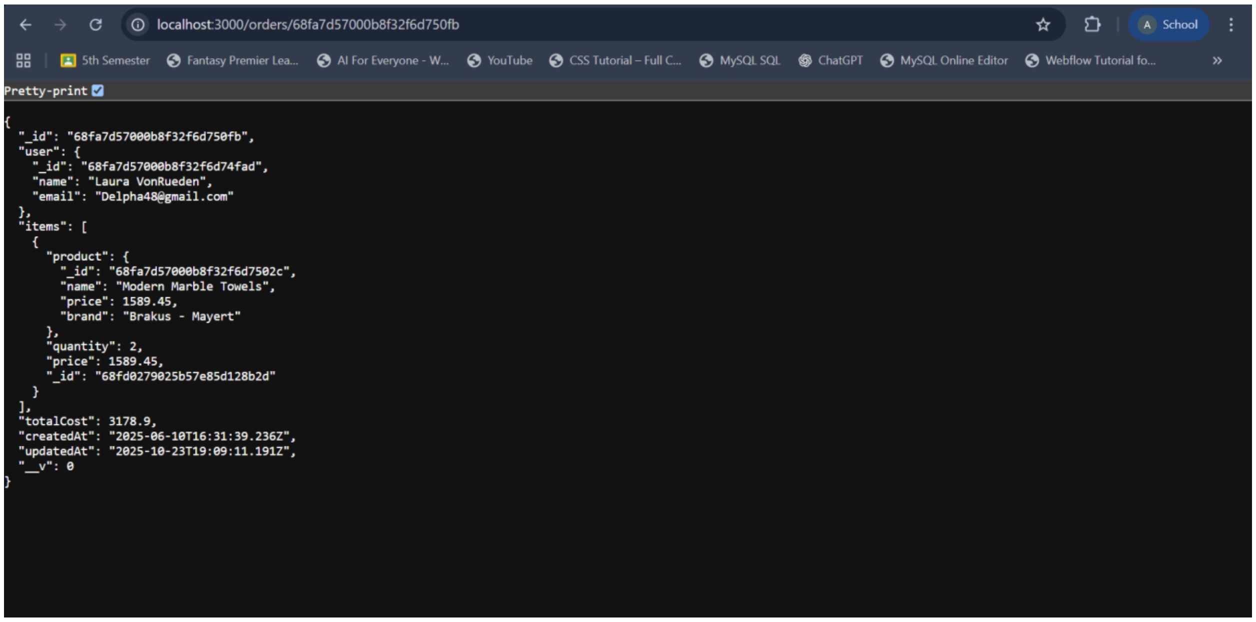
```
[
  {
    "_id": "68fa7d57000b8f32f6d7513d",
    "user": "68fa7d57000b8f32f6d74f9d",
    "items": [
      {
        "product": {
          "_id": "68fa7d57000b8f32f6d74fcf",
          "name": "Soft Silk Bacon",
          "price": 551.39
        },
        "quantity": 3,
        "price": 551.39,
        "_id": "68fd0279025b57e85d128c39"
      }
    ],
    "totalCost": 1654.17,
    "createdAt": "2025-10-10T09:57:58.333Z",
    "updatedAt": "2025-10-23T19:09:11.192Z",
    "__v": 0
  },
  {
    "_id": "68fa7d57000b8f32f6d75165",
    "user": "68fa7d57000b8f32f6d74f9d",
    "items": [
      {
        "product": {
          "_id": "68fa7d57000b8f32f6d75017",
          "name": "Modern Silk Mouse",
          "price": 1638.05
        },
        "quantity": 3,
        "price": 1638.05,
        "_id": "68fd0279025b57e85d128ce1"
```

2. Product Reviews: The GET /products/:id/reviews endpoint successfully fetches all reviews for a specific product.



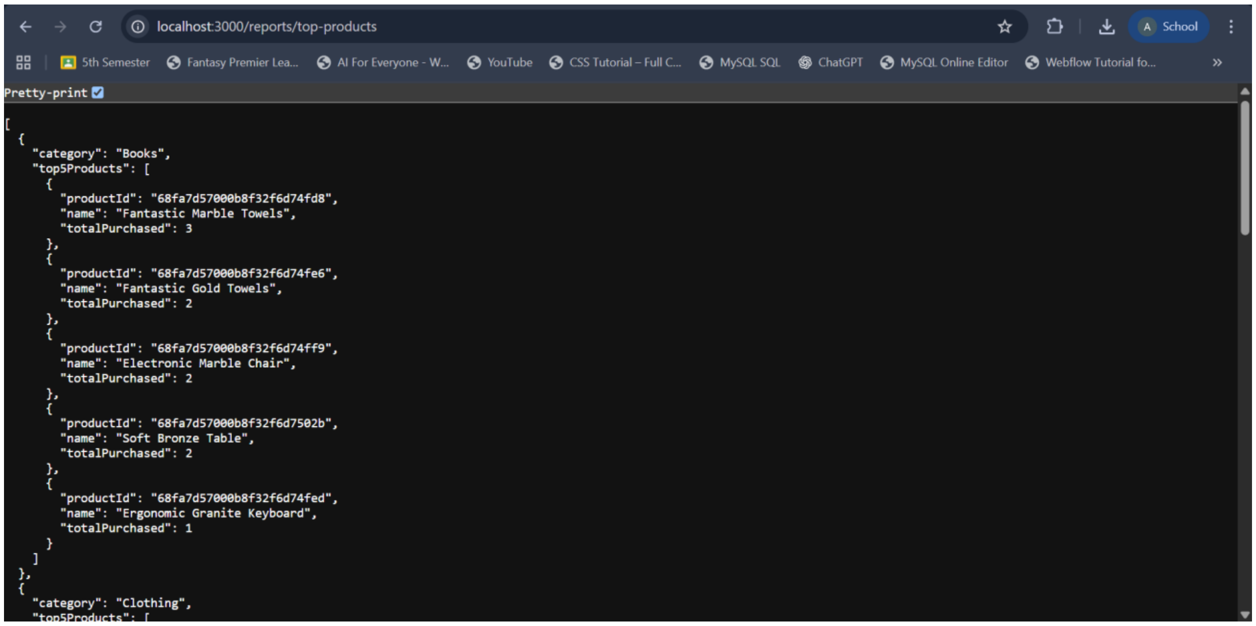
```
[
  {
    "_id": "68fa7d57000b8f32f6d750f3",
    "user": {
      "_id": "68fa7d57000b8f32f6d74fce",
      "name": "Alvin Walter",
      "location": "Abdielworth"
    },
    "product": "68fa7d57000b8f32f6d74fcf",
    "rating": 4,
    "text": "Saepe amoveo congregatio bibo. Caute carus accendo eum vester cum currus adipisci cohibeo universe. Conitor expedita adflicto atrocitas earum conicio teneo cresco
voluptatem.",
    "createdAt": "2025-08-20T14:19:23.499Z",
    "updatedAt": "2025-10-23T19:09:11.188Z",
    "__v": 0
  },
  {
    "_id": "68fa7d57000b8f32f6d750aa",
    "user": {
      "_id": "68fa7d57000b8f32f6d74fc7",
      "name": "Dr. Melinda Russel",
      "location": "West Ubaldofield"
    },
    "product": "68fa7d57000b8f32f6d74fcf",
    "rating": 2,
    "text": "Aegre rem defendo. Collum carmen tergeo. Clementia acsi aequitas utpote sperno curtus vester tantum volva aegre.",
    "createdAt": "2025-06-18T09:05:21.425Z",
    "updatedAt": "2025-10-23T19:09:11.185Z",
    "__v": 0
  },
  {
    "_id": "68fa7d57000b8f32f6d7508a",
    "user": {
      "_id": "68fa7d57000b8f32f6d74fbb"
```

3. Order Details: The GET /orders/:id endpoint successfully fetches the complete, populated details for a single order.



```
{
  "_id": "68fa7d57000b8f32f6d750fb",
  "user": {
    "_id": "68fa7d57000b8f32f6d74fad",
    "name": "Laura VonRueden",
    "email": "Delpha48@gmail.com"
  },
  "items": [
    {
      "product": {
        "_id": "68fa7d57000b8f32f6d7502c",
        "name": "Modern Marble Towels",
        "price": 1589.45,
        "brand": "Brakus - Mayert"
      },
      "quantity": 2,
      "price": 1589.45,
      "_id": "68fd0279025b57e85d128b2d"
    }
  ],
  "totalCost": 3178.9,
  "createdAt": "2025-06-10T16:31:39.236Z",
  "updatedAt": "2025-10-23T19:09:11.191Z",
  "__v": 0
}
```

4. Aggregation: The GET /reports/top-products endpoint correctly executes the aggregation pipeline.

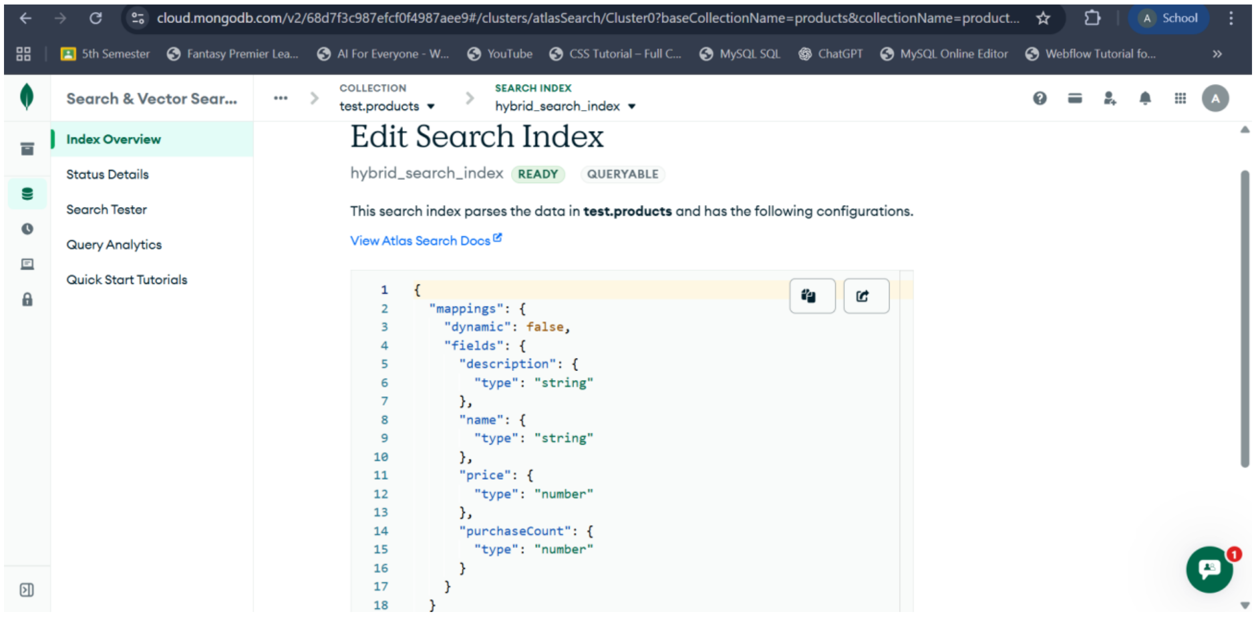


```
[
  {
    "category": "Books",
    "top5Products": [
      {
        "productId": "68fa7d57000b8f32f6d74fd8",
        "name": "Fantastic Marble Towels",
        "totalPurchased": 3
      },
      {
        "productId": "68fa7d57000b8f32f6d74fe6",
        "name": "Fantastic Gold Towels",
        "totalPurchased": 2
      },
      {
        "productId": "68fa7d57000b8f32f6d74ff9",
        "name": "Electronic Marble Chair",
        "totalPurchased": 2
      },
      {
        "productId": "68fa7d57000b8f32f6d7502b",
        "name": "Soft Bronze Table",
        "totalPurchased": 2
      },
      {
        "productId": "68fa7d57000b8f32f6d74fed",
        "name": "Ergonomic Granite Keyboard",
        "totalPurchased": 1
      }
    ]
  },
  {
    "category": "Clothing",
    "top5Products": [
```

Bonus: Hybrid Ranking Implementation

The bonus requirement for a hybrid search was successfully implemented using MongoDB Atlas Search.

Atlas Search Index: An index named hybrid_search_index was created on the products collection to enable "fuzzy" (similarity) search on name and description, as well as scoring on purchaseCount and price.



Edit Search Index

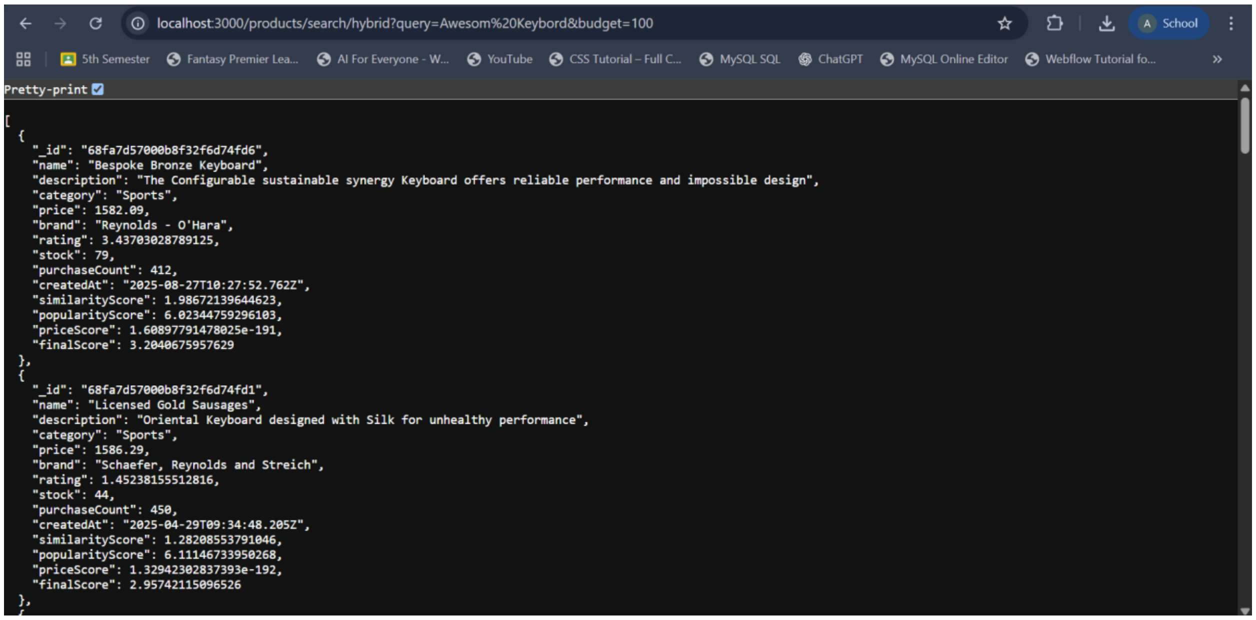
hybrid_search_index READY QUERYABLE

This search index parses the data in **test.products** and has the following configurations.

View Atlas Search Docs

```
{
  "mappings": {
    "dynamic": false,
    "fields": {
      "description": {
        "type": "string"
      },
      "name": {
        "type": "string"
      },
      "price": {
        "type": "number"
      },
      "purchaseCount": {
        "type": "number"
      }
    }
  }
```

5. Hybrid Search API: The GET /products/search/hybrid endpoint executes an aggregation pipeline that uses the $search operator. The screenshot below shows the API successfully finding a product with multiple misspellings (Awesom Keybord) and ranking the results based on similarity, popularity, and price.


localhost:3000/products/search/hybrid?query=Awesom%20Keybord&budget=100
Pretty-print ✓

```
[
  {
    "_id": "68fa7d57000b8f32f6d74fd6",
    "name": "Bespoke Bronze Keyboard",
    "description": "The Configurable sustainable synergy Keyboard offers reliable performance and impossible design",
    "category": "Sports",
    "price": 1582.09,
    "brand": "Reynolds - O'Hara",
    "rating": 3.43703028789125,
    "stock": 79,
    "purchaseCount": 412,
    "createdAt": "2025-08-27T10:27:52.762Z",
    "similarityScore": 1.98672139644623,
    "popularityScore": 6.02344759296103,
    "priceScore": 1.60897791478025e-191,
    "finalScore": 3.2040675957629
  },
  {
    "_id": "68fa7d57000b8f32f6d74fd1",
    "name": "Licensed Gold Sausages",
    "description": "Oriental Keyboard designed with Silk for unhealthy performance",
    "category": "Sports",
    "price": 1586.29,
    "brand": "Schaefer, Reynolds and Streich",
    "rating": 1.45238155512816,
    "stock": 44,
    "purchaseCount": 450,
    "createdAt": "2025-04-29T09:34:48.205Z",
    "similarityScore": 1.28208553791046,
    "popularityScore": 6.11146733950268,
    "priceScore": 1.32942302837393e-192,
    "finalScore": 2.95742115096526
  },
```

---

4. Conclusion

This project successfully demonstrated the end-to-end design and implementation of a scalable e-commerce backend. Key challenges included designing an efficient, hybrid schema (using both referencing and embedding) and implementing advanced query features.

The standard keyword search was fulfilled with a text index, and the complex aggregation pipeline efficiently derived business intelligence. Finally, by leveraging MongoDB Atlas Search, the bonus requirement for a powerful, weighted hybrid-ranking search was also successfully delivered, providing a modern, fault-tolerant search experience for the end-user