Task: Create Clever Strategies for a Retail Store Using Fake Data

Situation:

Imagine you're helping a small shop figure out some tricky problems. They need smart ideas to manage their stock and pricing.

What You Need to Do:

1. **Smart Restocking Plan**:

- Think of a smart way to decide how much of each product the shop should order. You'll use pretend sales data to figure this out.
- **Input Parameters**:
- salesData: JSON array containing past sales transactions. Each transaction should include:
- productID: ID of the product sold.
- quantitySold: Quantity of the product sold.
- timestamp: Timestamp of the sale.

Example: json

```
[
  {"productID": "123", "quantitySold": 10, "timestamp": "2024-06-01T10:00:00"},
  {"productID": "456", "quantitySold": 20, "timestamp": "2024-06-02T11:30:00"},
  {"productID": "123", "quantitySold": 15, "timestamp": "2024-06-03T09:45:00"}
]
```

- **Output Parameters**:
- restockPlan: JSON array containing recommendations for product restocking. Each recommendation should include:
- productID: ID of the product to restock.
- recommendedQuantity: Recommended quantity to reorder.

Example: json

```
[
  {"productID": "123", "recommendedQuantity": 50},
  {"productID": "456", "recommendedQuantity": 30}
]
```

2. **Clever Pricing Trick**:

- Come up with a clever trick to set prices that changes depending on what's happening in the pretend market. You'll use pretend prices from other shops and demand trends to help you.
- **Input Parameters**:
  - competitorPrices: JSON array containing prices of the same products from other shops.
  - demandTrends: JSON array containing demand trends for each product. Demand trends indicate whether the demand for a product is increasing, decreasing, or stable.

Example: json

```
[
 {"productID": "123", "price": 15},
 {"productID": "456", "price": 25}
]
```

Example: json

```
[
 {"productID": "123", "trend": "increasing"},
 {"productID": "456", "trend": "decreasing"}
]
```

- **Output Parameters**:
  - updatedPrices: JSON array containing updated prices for each product.

Example: json

```
[
 {"productID": "123", "updatedPrice": 20},
 {"productID": "456", "updatedPrice": 22}
]
```

3. Inventory Magic:

- Think of a magic way to decide how much of each product the shop should keep in stock. You'll use pretend info about what's popular and how long things last.
- **Input Parameters**:
  - popularityData: JSON array containing popularity data for each product. Popularity data indicates how popular each product is among customers.
    - Example:

      { "productID": "123", "popularityScore": 0.8 }

  - shelfLifeData: JSON array containing shelf life data for each product. Shelf life data indicates how long each product can be stored before it expires.
    - Example:

      { "productID": "123", "shelfLife": 30 } (30 days)

  - currentInventory: JSON array containing current inventory levels for each product.
    - Example:

      { "productID": "123", "currentStock": 100 }

- **Output Parameters**:
  - inventoryOptimization: JSON array containing recommended inventory adjustments for each product.
    - Example:

      { "productID": "123", "recommendedAdjustment": -20 }

      (Adjust inventory of product ID 123 by -20 units)

Include the following instructions regarding the README file:

1. **Setup Instructions**: Candidates should provide clear instructions on how to set up and run their code, including any prerequisites such as software dependencies or environment setup.
2. **Installation Steps**: Candidates should outline the necessary steps to install any required libraries or frameworks needed to run the code.
3. **Configuration**: If the code requires configuration settings or external resources (e.g., database connection strings, API keys), candidates should specify how to configure these settings.
4. **Running the Code**: Candidates should provide instructions on how to execute the code, including any command-line arguments or parameters that need to be provided.
5. **Input Data**: If applicable, candidates should specify how to provide input data to the code and any expected formats or conventions for the input data.
6. **Output**: Candidates should describe what the code produces as output and how to interpret or view the output results.
7. **Testing**: Candidates may include instructions for running tests or validating the code's functionality.
8. **Additional Information**: Any other important information or considerations that users should be aware of when running the code.

NOTES:

Optimization: Candidates should optimize their code for performance and efficiency, especially in algorithms and data processing tasks.

Code Structure: Encourage candidates to follow a clear and organized code structure, including modularization and separation of concerns.

Best Practices: Candidates should adhere to industry best practices and coding standards relevant to the programming language and framework being used.

Security: Emphasize the importance of writing secure code, including input validation, parameterized queries to prevent SQL injection, and protection against cross-site scripting (XSS) and other vulnerabilities.

Error Handling: Candidates should implement proper error handling mechanisms to gracefully handle unexpected scenarios and provide meaningful error messages to users.

Scalability: Candidates should design their solutions with scalability in mind, considering potential future growth and the ability to handle increased data volumes or user traffic.

Maintainability: Candidates should write code that is easy to maintain and extend, with clear naming conventions, consistent formatting, and minimal dependencies.