Automated Sales Order System

Be sure that you're a python expert with vast knowlege of pandas ai

How can i make sales order automated systems one impersonate as a unique customer and place order and another act as the sales order system of the company which recive order and send invoice to the customer.
Don't show code
just show how can i plan this efficiently and in scalabe format
if possible show flowcharts

Creating an automated sales order system with scalability and efficiency in mind requires proper planning. Below is a structured approach with steps and a flowchart for better understanding:

System Components

1. **Customer Emulator**:
   - Simulates customer behavior, including searching for products, adding to a cart, and placing orders.
   - Generates unique customer IDs for each transaction.
   - Sends order details to the company system via API.
2. **Sales Order System (Company Side)**:
   - Receives and validates orders.
   - Processes inventory checks.
   - Calculates invoices, taxes, and shipping.
   - Sends an invoice or order confirmation back to the customer emulator.
3. **Database**:
   - Central repository to store customer details, inventory data, sales orders, and invoices.

4. **API Gateway**:
    - Manages communication between the customer emulator and sales order system.
5. **Scalable Architecture**:
    - Use microservices to separate customer handling, order processing, and invoicing modules.

High-Level Workflow

1. **Customer Emulator Process**:
    - Generate unique customer ID.
    - Search and select products from inventory.
    - Place an order via API (HTTP POST).
2. **Sales Order System Process**:
    - Validate order details and customer ID.
    - Check product availability in inventory.
    - Generate invoice and confirm order.
    - Update inventory database.
3. **Communication**:
    - Use REST or GraphQL APIs for communication between customer emulator and sales order system.
4. **Scalability Considerations**:
    - Use a cloud-based database for scalability.
    - Implement caching for frequently accessed data.
    - Deploy on a distributed system with load balancers.

Flowchart

Customer Emulator Workflow

```csharp
[Start]
   ↓
[Generate Customer ID]
   ↓
[Search Products]
   ↓
[Select and Place Order]
   ↓
[Send Order via API]
   ↓
[Wait for Invoice/Response]
   ↓
[End]
```

Sales Order System Workflow

```csharp
[Start]
   ↓
[Receive Order via API]
   ↓
[Validate Order Details]
   ↓
[Check Inventory Availability]
   ↓
[Generate Invoice]
   ↓
[Send Invoice to Customer Emulator]
   ↓
[Update Inventory and Database]
   ↓
[End]
```

Scalable Format

1. **Database**:
   - Use a relational database (e.g., PostgreSQL) for structured data and NoSQL (e.g., MongoDB) for high scalability.
2. **Microservices**:
   - Separate services for order handling, inventory, and invoicing for modularity.
3. **Message Queues**:
   - Use queues (e.g., RabbitMQ, Kafka) for handling asynchronous tasks like invoice generation.
4. **Cloud Deployment**:
   - Deploy on AWS, Azure, or GCP for scalability.
   - Use serverless functions for specific tasks to reduce infrastructure overhead.
5. **Monitoring and Logging**:
   - Implement tools like ELK Stack or Prometheus for real-time monitoring and issue tracking.

Would you like me to create the flowchart as a visual diagram?

2/2

Both response is very useful , Now show come psudo code this

Here's a pseudocode representation of the system:

Customer Emulator

```plaintext
START CustomerEmulator

# Step 1: Generate Unique Customer ID
customer_id = generate_unique_customer_id()
```

```
# Step 2: Browse Products
product_list = fetch_products_from_inventory()

# Step 3: Select Products
selected_products = select_products(product_list)

# Step 4: Place Order
order_details = {
    "customer_id": customer_id,
    "products": selected_products
}
response = send_order_to_company_api(order_details)

# Step 5: Handle Response
IF response.status == "SUCCESS":
    print("Order placed successfully!")
    print("Invoice Details:", response.invoice)
ELSE:
    print("Failed to place order. Reason:", response.error)

END CustomerEmulator
```

Sales Order System

plaintext

```
START SalesOrderSystem

# Step 1: Receive Order via API
order = receive_order_request()

# Step 2: Validate Order
IF NOT validate_order(order):
    return {"status": "FAILURE", "error": "Invalid Order"}

# Step 3: Check Inventory
available = check_inventory(order.products)
IF NOT available:
```

```
    return {"status": "FAILURE", "error": "Product Out of Stock"}

# Step 4: Generate Invoice
invoice = generate_invoice(order)

# Step 5: Send Invoice to Customer
send_invoice_to_customer(order.customer_id, invoice)

# Step 6: Update Inventory and Database
update_inventory(order.products)
save_order_to_database(order, invoice)

RETURN {"status": "SUCCESS", "invoice": invoice}

END SalesOrderSystem
```

Helper Functions

1. **Customer Emulator**:

   ```plaintext
   generate_unique_customer_id():
       return "CUST" + random_number()

   fetch_products_from_inventory():
       return api_call("/inventory")

   select_products(product_list):
       return choose_random_items(product_list)

   send_order_to_company_api(order_details):
       return api_post("/place_order", order_details)
   ```

2. **Sales Order System**:

   ```plaintext
   ```

```
receive_order_request():
    return api_receive_post_request()

validate_order(order):
    return all([
        order.customer_id != None,
        order.products != None
    ])

check_inventory(products):
    FOR product IN products:
        IF inventory[product.id].quantity < product.quantity:
            return False
    return True

generate_invoice(order):
    total = 0
    FOR product IN order.products:
        total += product.price * product.quantity
    return {"total": total, "order_id": generate_order_id()}

send_invoice_to_customer(customer_id, invoice):
    return api_call("/send_invoice", {"customer_id": customer_id, "invoice":
invoice})

update_inventory(products):
    FOR product IN products:
        inventory[product.id].quantity -= product.quantity

save_order_to_database(order, invoice):
    database.orders.insert(order)
    database.invoices.insert(invoice)
```

This pseudocode gives a clear structure for implementing the system. It’s modular and scalable, with reusable functions for each process. Let me know if you'd like me to refine any specific part or expand on the architecture!

How to make fastest API for order system in robust and scalable format , such that if million users comes in it can be easily managed

To create a **fast, robust, and scalable API** capable of handling a million users, you need to focus on high-performance architecture, efficient code, and a scalable deployment strategy. Here's how you can achieve this:

---

1. Design a Scalable Architecture

Microservices

- Split the system into independent services:
  - **Order Service**: Handles incoming orders.
  - **Inventory Service**: Manages stock availability.
  - **Invoice Service**: Processes payments and generates invoices.
  - **Notification Service**: Sends order confirmations and invoices.

API Gateway

- Use an API Gateway (e.g., AWS API Gateway, Kong) to route requests to the appropriate service.
- Provides rate-limiting, caching, and security.

Database Sharding

- Distribute the database across multiple nodes (shards) to handle high traffic.
- Use customer IDs or region-based partitioning to direct queries to specific shards.

Load Balancing

- Use load balancers (e.g., AWS ELB, NGINX) to distribute traffic across servers.
- Deploy services in multiple regions for global coverage.

Message Queues

- For asynchronous tasks like invoice generation or inventory updates, use message queues (e.g., RabbitMQ, Kafka).

- Prevents bottlenecks in the order API.

Caching

- Use caching (e.g., Redis, Memcached) to store frequently accessed data like inventory or product details.

---

2. Use High-Performance Frameworks and Technologies

Frameworks

- Choose lightweight frameworks like **FastAPI** or **Flask** for Python.
- For extreme performance, consider **Go** or **Node.js** for API development.

Asynchronous Programming

- Use asynchronous patterns (e.g., `asyncio` in Python) to handle multiple requests concurrently.

Database Optimization

- Use connection pooling to efficiently manage database connections.
- Index frequently queried fields.
- Optimize SQL queries or use NoSQL (e.g., MongoDB) for unstructured data.

Horizontal Scaling

- Design stateless APIs so they can scale horizontally.
- Use Kubernetes or Docker Swarm for container orchestration.

---

3. Implement Scalability and Fault Tolerance

Auto-Scaling

- Configure auto-scaling groups to spin up new instances when traffic spikes.
- Use services like AWS Lambda for serverless functions to handle specific tasks dynamically.

Distributed Systems

- Deploy your services across multiple data centers and regions.
- Use a Content Delivery Network (CDN) for static assets.

Rate Limiting and Throttling

- Prevent abuse by limiting the number of requests per user/IP using tools like **API Gateway** or custom middleware.

---

4. Monitoring and Logging

Real-Time Monitoring

- Use tools like Prometheus, Grafana, or AWS CloudWatch to monitor system performance.
- Set up alerts for critical metrics like latency, error rates, and server utilization.

Centralized Logging

- Use tools like ELK Stack (Elasticsearch, Logstash, Kibana) for centralized logging and debugging.

---

5. API Endpoint Design

Efficient Endpoints

- Design RESTful or GraphQL APIs with clear and concise endpoints.
- Example:
  - **POST /order**: Place a new order.
  - **GET /order/{order_id}**: Fetch order details.
  - **GET /products**: Fetch product list.

Minimal Payload

- Use efficient data serialization formats like **Protocol Buffers** (Protobuf) or **MessagePack** instead of JSON.

Versioning

- Implement API versioning (e.g., `/v1/order`) to ensure backward compatibility.

---

6. Deployment Strategy

Containerization

- Use Docker to containerize your application for consistent deployment.

Orchestration

- Use Kubernetes or AWS ECS for managing containers and scaling.

CD/CI Pipeline

- Automate deployment using tools like Jenkins, GitHub Actions, or GitLab CI/CD.

Workflow Diagram for High-Performance API

Request Flow

```scss
[Client]
   ↓
[API Gateway] -- (Rate Limiting, Authentication)
   ↓
[Order Microservice]
   ↓        ↘
[Inventory Service] [Invoice Service] -- (Queue for Async Tasks)
   ↓                        ↘
[Database Cluster]    [Notification Service]
```

Key Tips for High Performance

1. **Minimize Latency**:
   - Deploy services closer to users with edge computing.
   - Optimize database queries and reduce network calls.
2. **Fault Tolerance**:
   - Use circuit breakers and retries for API calls.
   - Implement data replication in databases.
3. **Test Under Load**:
   - Use tools like JMeter or Locust for load testing.

By following this plan, you can build a fast and scalable API ready to handle millions of concurrent users effectively. Let me know if you'd like to explore specific parts further!

show a the python code for order system API , we'll do the Order system first then Customer side don't put any details about that

‹/› Order System Api

The provided code implements the **Order System API** using FastAPI. It supports the following functionalities:

1. **Place an Order**:
   - Validates the order for inventory availability.
   - Updates the inventory upon successful order placement.
   - Calculates the total price and returns an invoice.
2. **Check Inventory**:
   - Allows retrieval of current inventory details.

This API is ready to be tested. Let me know if you need further extensions, such as integrating a database or adding error handling!

How the backed for inventory would be managed ?

Managing the backend for inventory involves choosing a reliable and scalable storage solution, implementing efficient database models, and setting up processes to handle inventory updates and queries. Here's how you can manage the backend for inventory:

---

1. Database Choice

Relational Database (SQL):

- Suitable for structured inventory data.
- Use **PostgreSQL** or **MySQL** for transactional consistency.
- Ideal if your inventory needs features like complex queries, relationships, and ACID compliance.

NoSQL Database:

- For high scalability and flexibility, use NoSQL databases like **MongoDB** or **DynamoDB**.
- Suitable for real-time updates and distributed systems.

---

2. Inventory Schema Design

SQL Example:

```sql
CREATE TABLE Inventory (
    product_id VARCHAR(36) PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    quantity INT NOT NULL,
    last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

NoSQL Example (MongoDB):

```json
{
    "product_id": "item1",
```

```json
    "name": "Product A",
    "price": 50.00,
    "quantity": 100,
    "last_updated": "2024-12-24T10:00:00Z"
}
```

---

3. Backend Implementation

CRUD Operations:

- **Create**: Add new products to the inventory.
- **Read**: Fetch inventory details for order placement.
- **Update**: Update quantities after an order is placed.
- **Delete**: Remove discontinued products.

Transactions:

- Use database transactions to ensure atomicity, especially for operations like checking stock and updating quantities.

---

4. Integrating Inventory in the API

1. **Database Connection**:
    - Use an ORM like **SQLAlchemy** (SQL) or **Pymongo** (NoSQL) in Python.
2. **Inventory Service**:
    - Abstract inventory operations into a dedicated microservice or module.
    - Expose inventory endpoints like:
        - `/inventory`: Fetch all products.
        - `/inventory/{product_id}`: Fetch details of a single product.
        - `/inventory/update`: Update stock levels.

---

5. Handling High Traffic

Caching:

- Use **Redis** or **Memcached** to cache frequently accessed inventory data.
- Sync cache with the database periodically.

Partitioning and Sharding:

- Partition the database by product category or region to reduce load.
- Shard inventory data for horizontal scaling.

Load Balancing:

- Distribute database queries across replicas using a load balancer.

---

6. Real-Time Updates

- Implement **event-driven systems** using tools like **Kafka** or **RabbitMQ**.
- Update inventory asynchronously to handle high-volume transactions.

---

7. Monitoring and Logging

- Use monitoring tools like **Prometheus** or **Datadog** to track inventory updates and database performance.
- Log all inventory changes for auditing purposes.

Would you like me to provide a specific example of integrating a database (e.g., PostgreSQL) with the current inventory API?

Yes provide some integration and with a free to use database

Order System Api

```
from fastapi import FastAPI, HTTPException, Depends
from pydantic import BaseModel
from typing import List
import uuid
from sqlalchemy import create_engine, Column, String, Integer, Float, Table, MetaData
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

# Initialize FastAPI app
app = FastAPI()

# Database setup
DATABASE_URL = "sqlite:///./inventory.db"  # Free-to-use SQLite database
engine = create_engine(DATABASE_URL, connect_args={"check_same_thread": False})
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

# Inventory Table
class Inventory(Base):
    __tablename__ = "inventory"

    product_id = Column(String, primary_key=True, index=True)
    name = Column(String, nullable=False)
    price = Column(Float, nullable=False)
    quantity = Column(Integer, nullable=False)

# Create tables
Base.metadata.create_all(bind=engine)

# Dependency to get the database session
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

# Models
class OrderItem(BaseModel):
    product_id: str
    quantity: int

class Order(BaseModel):
    customer_id: str
    items: List[OrderItem]
```

```python
class Invoice(BaseModel):
    order_id: str
    total: float

class Product(BaseModel):
    product_id: str
    name: str
    price: float
    quantity: int

# Helper Functions
def validate_order_items(items: List[OrderItem], db):
    for item in items:
        product = db.query(Inventory).filter(Inventory.product_id == item.product_id).f
        if not product:
            raise HTTPException(status_code=404, detail=f"Product {item.product_id} not
        if product.quantity < item.quantity:
            raise HTTPException(
                status_code=400, detail=f"Insufficient stock for product {item.product_
            )

def update_inventory(items: List[OrderItem], db):
    for item in items:
        product = db.query(Inventory).filter(Inventory.product_id == item.product_id).f
        product.quantity -= item.quantity
        db.commit()

# API Endpoints
@app.post("/place_order", response_model=Invoice)
def place_order(order: Order, db: SessionLocal = Depends(get_db)):
    # Validate order items
    validate_order_items(order.items, db)

    # Calculate total
    total = 0
    for item in order.items:
        product = db.query(Inventory).filter(Inventory.product_id == item.product_id).f
        total += product.price * item.quantity

    # Generate order ID
    order_id = str(uuid.uuid4())

    # Update inventory
    update_inventory(order.items, db)

    # Return invoice
    return Invoice(order_id=order_id, total=total)

@app.get("/inventory", response_model=List[Product])
def get_inventory(db: SessionLocal = Depends(get_db)):
    products = db.query(Inventory).all()
```

```
    return [Product(product_id=p.product_id, name=p.name, price=p.price, quantity=p.qua

@app.post("/inventory/add", response_model=Product)
def add_product(product: Product, db: SessionLocal = Depends(get_db)):
    new_product = Inventory(
        product_id=product.product_id,
        name=product.name,
        price=product.price,
        quantity=product.quantity
    )
    db.add(new_product)
    db.commit()
    db.refresh(new_product)
    return product
```

The updated code now integrates an SQLite database (a free-to-use database) for managing inventory. Here's how the changes enhance the application:

1. **Database Integration**:
   - SQLite is used to store inventory data persistently.
   - SQLAlchemy ORM manages database operations seamlessly.
2. **Endpoints for Inventory Management**:
   - **GET /inventory**: Fetches all inventory items.
   - **POST /inventory/add**: Allows adding new products to the inventory.
3. **Dynamic Inventory Updates**:
   - Inventory quantities are updated in the database when orders are placed.

This setup provides a robust foundation for your order system, with the flexibility to switch to a more scalable database (e.g., PostgreSQL or MySQL) later. Let me know if you need further customization!