Part 1) Code Review & Debugging Analysis

1. Issues Identified

- **Critical Issues:**

1. **No input validation or error handling**
    - Missing try-catch blocks for database operations
    - No validation for required fields
    - No type checking for incoming data
2. **Missing SKU uniqueness constraint enforcement**
    - Code doesn't check if SKU already exists before creating product
    - Could lead to duplicate SKUs despite business requirement
3. **No transaction management**
    - Two separate commits create a race condition
    - If inventory creation fails, product is already committed (orphaned product)
4. **Improper HTTP status codes**
    - Returns 200 by default even on successful creation (should be 201)
    - No error status codes for failures
5. **Security vulnerability: No authentication/authorization**
    - No check if user has permission to create products
    - No verification of warehouse_id ownership
6. **Data integrity issues:**
    - No validation that warehouse_id exists
    - No constraints on price (could be negative)

- No constraints on initial_quantity (could be negative)

Design Issues:

7. **Poor error responses**
    - Generic error messages won't help API consumers debug issues
8. **Missing audit trail**
    - No timestamp, created_by, or modification tracking
9. **Potential duplicate inventory records**
    - If product already exists in warehouse, creates duplicate inventory entry

2) Impact Analysis

Issue	Production Impact
No error handling	**API crashes expose stack traces, potential security risk. Users get 500 errors with no context.**
Missing SKU validation	**Duplicate products created, inventory becomes inconsistent, reporting breaks.**
Split transactions	**Orphaned products without inventory records. Data corruption. Cleanup requires manual intervention.**
No auth	**Any user can create products for any warehouse. Data breach potential.**
No warehouse validation	**Foreign key violations or products linked to non-existent warehouses.**
Negative values allowed	**Business logic violations: -10 products, -50 inventory count.**

Issue	Production Impact
Poor error messages	**Support tickets increase , integration partners can't debug issues.**
No audit trail	**Can't track who created products, regulatory compliance issues.**

3) corrected version :

```
@app.route('/api/products', methods=['POST'])
@require_auth
def create_product(current_user):
    """Create a new product with initial inventory.
    Required fields: name, sku, price, warehouse_id, initial_quantity
    Returns: 201 on success, 400 on validation error, 409 on conflict
    """
    try:
        data = request.get_json()
```

```
# Validate required fields
required_fields = ['name', 'sku', 'price', 'warehouse_id', 'initial_quantity']
missing_fields = [field for field in required_fields if field not in data]

if missing_fields:
    return jsonify({
        "error": "Missing required fields",
        "missing_fields": missing_fields
    }), 400

# Validate data types and business rules
try:
    price = Decimal(str(data['price']))
    if price < 0:
        return jsonify({"error": "Price cannot be negative"}), 400
except (InvalidOperation, ValueError):
    return jsonify({"error": "Invalid price format"}), 400

try:
    initial_quantity = int(data['initial_quantity'])
    if initial_quantity < 0:
        return jsonify({"error": "Initial quantity cannot be negative"}), 400
except (ValueError, TypeError):
    return jsonify({"error": "Invalid quantity format"}), 400

# Validate warehouse exists and user has access
```

```
warehouse = Warehouse.query.get(data['warehouse_id'])
if not warehouse:
    return jsonify({"error": "Warehouse not found"}), 404

if not current_user.has_access_to_warehouse(warehouse.id):
    return jsonify({"error": "Access denied to this warehouse"}), 403

# Check SKU uniqueness (platform-wide)
existing_product = Product.query.filter_by(sku=data['sku']).first()
if existing_product:
    return jsonify({
        "error": "SKU already exists",
        "existing_product_id": existing_product.id
    }), 409

# Begin atomic transaction
try:
    # Create product
    product = Product(
        name=data['name'].strip(),
        sku=data['sku'].strip().upper(),  # Normalize SKU
        price=price,
        created_by=current_user.id,
        created_at=datetime.utcnow()
    )
    db.session.add(product)
```

```
db.session.flush()  # Get product.id without committing

# Check if inventory already exists for this warehouse
existing_inventory = Inventory.query.filter_by(
    product_id=product.id,
    warehouse_id=data['warehouse_id']
).first()

if existing_inventory:
    # Update existing inventory
    existing_inventory.quantity += initial_quantity
    existing_inventory.updated_at = datetime.utcnow()
    existing_inventory.updated_by = current_user.id
else:
    # Create new inventory record
    inventory = Inventory(
        product_id=product.id,
        warehouse_id=data['warehouse_id'],
        quantity=initial_quantity,
        created_by=current_user.id,
        created_at=datetime.utcnow()
    )
    db.session.add(inventory

# Commit everything together
db.session.commit()
```

```
        return jsonify({
            "message": "Product created successfully",
            "product": {
                "id": product.id,
                "name": product.name,
                "sku": product.sku,
                "price": str(product.price),
                "warehouse_id": data['warehouse_id'],
                "initial_quantity": initial_quantity
            }
        }), 201


except IntegrityError as e:
        db.session.rollback()
        # Handle database constraint violations
        return jsonify({
            "error": "Database constraint violation",
            "details": str(e.orig)
        }), 409

    except Exception as e:
        db.session.rollback()
        app.logger.error(f"Unexpected error in create_product: {str(e)}")
        return jsonify({
```

```
        "error": "An unexpected error occurred",
        "message": "Please contact support"
    }), 500
```

Key Improvements Explained

1. **Transaction Safety:** Single commit with rollback on any failure
2. **Input Validation:** Type checking, business rule validation, required field checks
3. **Security:** Authentication, authorization, warehouse access control
4. **Error Handling:** Comprehensive try-catch with appropriate HTTP status codes
5. **Data Integrity:** SKU uniqueness check, warehouse existence validation
6. **Audit Trail:** created_by, created_at timestamps
7. **Better UX**: Descriptive error messages for API consumers
8. **Idempotency consideration:** Checks for existing inventory records

Part 2: Database Design for StockFlow

1. Database Schema Design :

```
-- ============================================
-- ORGANIZATIONS & USER MANAGEMENT
-- ============================================

CREATE TABLE organizations (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    slug VARCHAR(100) UNIQUE NOT NULL,
    subscription_tier VARCHAR(50) DEFAULT 'free',
```

```
  sku_case_sensitive BOOLEAN DEFAULT FALSE,
  created_at TIMESTAMP NOT NULL DEFAULT NOW(),
  updated_at TIMESTAMP NOT NULL DEFAULT NOW(),
  is_active BOOLEAN DEFAULT TRUE
);

CREATE INDEX idx_organizations_slug ON organizations(slug);
CREATE INDEX idx_organizations_active ON organizations(is_active);

CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  organization_id INTEGER NOT NULL REFERENCES organizations(id),
  email VARCHAR(255) NOT NULL UNIQUE,
  password_hash VARCHAR(255) NOT NULL,
  first_name VARCHAR(100),
  last_name VARCHAR(100),
  role VARCHAR(50) NOT NULL DEFAULT 'user', -- admin, manager, user, viewer
  is_active BOOLEAN DEFAULT TRUE,
  last_login_at TIMESTAMP,
  created_at TIMESTAMP NOT NULL DEFAULT NOW(),
  updated_at TIMESTAMP NOT NULL DEFAULT NOW()
);

CREATE INDEX idx_users_org_id ON users(organization_id);
CREATE INDEX idx_users_email ON users(email);
CREATE INDEX idx_users_active ON users(organization_id, is_active);
```

```
-- ========================================
-- WAREHOUSE MANAGEMENT
-- ========================================

CREATE TABLE warehouses (
    id SERIAL PRIMARY KEY,
    organization_id INTEGER NOT NULL REFERENCES organizations(id),
    name VARCHAR(255) NOT NULL,
    code VARCHAR(50), -- Short identifier like "WH-NYC-01"
    address_line1 VARCHAR(255),
    address_line2 VARCHAR(255),
    city VARCHAR(100),
    state_province VARCHAR(100),
    postal_code VARCHAR(20),
    country VARCHAR(2), -- ISO country code
    contact_name VARCHAR(100),
    contact_phone VARCHAR(20),
    contact_email VARCHAR(255),
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW(),
    created_by INTEGER REFERENCES users(id),

    CONSTRAINT unique_warehouse_code_per_org UNIQUE(organization_id, code)
);

CREATE INDEX idx_warehouses_org_id ON warehouses(organization_id);
```

```
CREATE INDEX idx_warehouses_active ON warehouses(organization_id, is_active);


-- ============================================
-- SUPPLIER MANAGEMENT
-- ============================================


CREATE TABLE suppliers (
    id SERIAL PRIMARY KEY,
    organization_id INTEGER NOT NULL REFERENCES organizations(id),
    name VARCHAR(255) NOT NULL,
    code VARCHAR(50), -- Internal supplier code
    contact_name VARCHAR(100),
    contact_email VARCHAR(255),
    contact_phone VARCHAR(20),
    address_line1 VARCHAR(255),
    address_line2 VARCHAR(255),
    city VARCHAR(100),
    state_province VARCHAR(100),
    postal_code VARCHAR(20),
    country VARCHAR(2),
    payment_terms VARCHAR(100), -- "Net 30", "Net 60", etc.
    currency VARCHAR(3) DEFAULT 'USD', -- ISO currency code
    tax_id VARCHAR(50), -- For tax/compliance
    notes TEXT,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW(),
```

```
    created_by INTEGER REFERENCES users(id),

    CONSTRAINT unique_supplier_code_per_org UNIQUE(organization_id, code)
);

CREATE INDEX idx_suppliers_org_id ON suppliers(organization_id);
CREATE INDEX idx_suppliers_active ON suppliers(organization_id, is_active);

-- ============================================
-- PRODUCT MANAGEMENT
-- ============================================

CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    organization_id INTEGER NOT NULL REFERENCES organizations(id),
    sku VARCHAR(100) NOT NULL, -- Original SKU as entered
    sku_normalized VARCHAR(100) NOT NULL, -- Uppercase, no spaces for uniqueness
    name VARCHAR(255) NOT NULL,
    description TEXT,
    product_type VARCHAR(50) DEFAULT 'simple', -- simple, bundle, variant
    category VARCHAR(100), -- Electronics, Apparel, etc.
    unit_of_measure VARCHAR(20) DEFAULT 'each', -- each, box, pallet, kg, lb

    -- Pricing
    cost_price DECIMAL(15, 4), -- What we pay supplier
    sale_price DECIMAL(15, 4), -- What we sell for
    currency VARCHAR(3) DEFAULT 'USD',
```

```
  -- Physical attributes
  weight DECIMAL(10, 4),
  weight_unit VARCHAR(10), -- kg, lb, g
  length DECIMAL(10, 2),
  width DECIMAL(10, 2),
  height DECIMAL(10, 2),
  dimension_unit VARCHAR(10), -- cm, in, m

  -- Inventory settings
  low_stock_threshold INTEGER DEFAULT 10,
  reorder_point INTEGER,
  reorder_quantity INTEGER,



  -- Soft delete and audit
  is_deleted BOOLEAN DEFAULT FALSE,
  deleted_at TIMESTAMP,
  deleted_by INTEGER REFERENCES users(id),
  created_at TIMESTAMP NOT NULL DEFAULT NOW(),
  updated_at TIMESTAMP NOT NULL DEFAULT NOW(),
  created_by INTEGER REFERENCES users(id),
  updated_by INTEGER REFERENCES users(id),

  CONSTRAINT unique_sku_per_org UNIQUE(organization_id, sku_normalized)
);
```

```
CREATE INDEX idx_products_org_id ON products(organization_id);

CREATE INDEX idx_products_sku ON products(organization_id, sku_normalized);

CREATE INDEX idx_products_active ON products(organization_id, is_deleted) WHERE is_deleted = FALSE;

CREATE INDEX idx_products_type ON products(product_type);

CREATE INDEX idx_products_category ON products(organization_id, category);

CREATE INDEX idx_products_barcode ON products(barcode) WHERE barcode IS NOT NULL;


-- Full-text search on product name and description

CREATE INDEX idx_products_search ON products USING GIN(

  to_tsvector('english', COALESCE(name, '') || ' ' || COALESCE(description, ''))

);
```

```
-- ============================================

-- PRODUCT-SUPPLIER RELATIONSHIPS

-- ============================================


CREATE TABLE product_suppliers (

  id SERIAL PRIMARY KEY,

  product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE CASCADE,

  supplier_id INTEGER NOT NULL REFERENCES suppliers(id) ON DELETE CASCADE,

  supplier_sku VARCHAR(100), -- Supplier's SKU for this product

  cost_price DECIMAL(15, 4), -- Cost from this specific supplier

  lead_time_days INTEGER, -- How long to get stock from supplier

  minimum_order_quantity INTEGER,
```

```
  is_preferred BOOLEAN DEFAULT FALSE, -- Preferred supplier for this product
  notes TEXT,
  created_at TIMESTAMP NOT NULL DEFAULT NOW(),
  updated_at TIMESTAMP NOT NULL DEFAULT NOW(),

  CONSTRAINT unique_product_supplier UNIQUE(product_id, supplier_id)
);

CREATE INDEX idx_product_suppliers_product ON product_suppliers(product_id);
CREATE INDEX idx_product_suppliers_supplier ON product_suppliers(supplier_id);
CREATE INDEX idx_product_suppliers_preferred ON product_suppliers(product_id, is_preferred)
  WHERE is_preferred = TRUE;



-- ============================================
-- PRODUCT BUNDLES (Bill of Materials)
-- ============================================

CREATE TABLE product_bundles (
  id SERIAL PRIMARY KEY,
  bundle_product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE CASCADE,
  component_product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE CASCADE,
  quantity DECIMAL(10, 4) NOT NULL DEFAULT 1, -- Quantity of component in bundle
  created_at TIMESTAMP NOT NULL DEFAULT NOW(),

  -- Prevent circular dependencies
```

```
  CONSTRAINT no_self_reference CHECK (bundle_product_id != component_product_id),
  CONSTRAINT unique_bundle_component UNIQUE(bundle_product_id, component_product_id)
);

CREATE INDEX idx_bundles_bundle_id ON product_bundles(bundle_product_id);
CREATE INDEX idx_bundles_component_id ON product_bundles(component_product_id);

-- ============================================
-- INVENTORY MANAGEMENT
-- ============================================

CREATE TABLE inventory (
  id SERIAL PRIMARY KEY,
  product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE CASCADE,
  warehouse_id INTEGER NOT NULL REFERENCES warehouses(id) ON DELETE CASCADE,

  -- Quantities
  quantity_available INTEGER NOT NULL DEFAULT 0, -- Available for sale
  quantity_reserved INTEGER NOT NULL DEFAULT 0, -- Reserved for orders
  quantity_damaged INTEGER NOT NULL DEFAULT 0, -- Damaged/unsellable
  quantity_in_transit INTEGER NOT NULL DEFAULT 0, -- Being transferred

  -- Calculated: Total physical stock
  -- quantity_on_hand = quantity_available + quantity_reserved + quantity_damaged
  -- Timestamps
  last_counted_at TIMESTAMP, -- Last physical count
  last_counted_by INTEGER REFERENCES users(id),
```

```
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW(),

    CONSTRAINT unique_product_warehouse UNIQUE(product_id, warehouse_id),
    CONSTRAINT non_negative_available CHECK (quantity_available >= 0),
    CONSTRAINT non_negative_reserved CHECK (quantity_reserved >= 0),
    CONSTRAINT non_negative_damaged CHECK (quantity_damaged >= 0)
);

CREATE INDEX idx_inventory_product ON inventory(product_id);
CREATE INDEX idx_inventory_warehouse ON inventory(warehouse_id);
CREATE INDEX idx_inventory_low_stock ON inventory(product_id, warehouse_id,
quantity_available);
```

-- View for total available inventory across all warehouses

```
CREATE VIEW product_total_inventory AS
SELECT
    product_id,
    SUM(quantity_available) as total_available,
    SUM(quantity_reserved) as total_reserved,
    SUM(quantity_damaged) as total_damaged,
    SUM(quantity_available + quantity_reserved + quantity_damaged) as total_on_hand
FROM inventory
GROUP BY product_id;
```

-- ==========

-- INVENTORY TRANSACTION HISTORY

-- ==========

```
CREATE TABLE inventory_transactions (
    id BIGSERIAL PRIMARY KEY, -- BIGSERIAL for high volume
    product_id INTEGER NOT NULL REFERENCES products(id),
    warehouse_id INTEGER NOT NULL REFERENCES warehouses(id),

    -- Transaction details
    transaction_type VARCHAR(50) NOT NULL,
    -- Types: purchase, sale, adjustment, transfer_in, transfer_out,
    --        return, damage, recount, manufacturing

    quantity_change INTEGER NOT NULL, -- Positive or negative
    quantity_before INTEGER NOT NULL,
    quantity_after INTEGER NOT NULL,

    -- Cost tracking
    unit_cost DECIMAL(15, 4),
    total_cost DECIMAL(15, 4),

    -- Reference data
    reference_type VARCHAR(50), -- purchase_order, sales_order, transfer, adjustment
    reference_id INTEGER, -- ID of related record
    -- Audit
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    created_by INTEGER NOT NULL REFERENCES users(id)
```

```
);

CREATE INDEX idx_inv_trans_product ON inventory_transactions(product_id);
CREATE INDEX idx_inv_trans_warehouse ON inventory_transactions(warehouse_id);
CREATE INDEX idx_inv_trans_type ON inventory_transactions(transaction_type);
CREATE INDEX idx_inv_trans_created ON inventory_transactions(created_at DESC);
CREATE INDEX idx_inv_trans_reference ON inventory_transactions(reference_type, reference_id);




-- ============================================
-- PURCHASE ORDERS (for supplier restocking)
-- ============================================

CREATE TABLE purchase_orders (
  id SERIAL PRIMARY KEY,
  organization_id INTEGER NOT NULL REFERENCES organizations(id),
  supplier_id INTEGER NOT NULL REFERENCES suppliers(id),
  warehouse_id INTEGER NOT NULL REFERENCES warehouses(id), -- Destination warehouse

  po_number VARCHAR(50) NOT NULL, -- Human-readable PO number
  status VARCHAR(50) NOT NULL DEFAULT 'draft',
```

```
    -- draft, submitted, approved, partially_received, received, cancelled

    order_date DATE NOT NULL,
    expected_delivery_date DATE,
    actual_delivery_date DATE,

    -- Totals
    subtotal DECIMAL(15, 4),
    tax_amount DECIMAL(15, 4),
    shipping_cost DECIMAL(15, 4),
    total_amount DECIMAL(15, 4),
    currency VARCHAR(3) DEFAULT 'USD',

    notes TEXT,
    terms TEXT, -- Payment terms, delivery terms

    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW(),
    created_by INTEGER REFERENCES users(id),
    approved_by INTEGER REFERENCES users(id),
    approved_at TIMESTAMP,

    CONSTRAINT unique_po_number_per_org UNIQUE(organization_id, po_number)
);

CREATE INDEX idx_po_org_id ON purchase_orders(organization_id);
CREATE INDEX idx_po_supplier ON purchase_orders(supplier_id);
```

```
CREATE INDEX idx_po_status ON purchase_orders(status);
CREATE INDEX idx_po_dates ON purchase_orders(order_date, expected_delivery_date);

CREATE TABLE purchase_order_items (
  id SERIAL PRIMARY KEY,
  purchase_order_id INTEGER NOT NULL REFERENCES purchase_orders(id) ON DELETE CASCADE,
  product_id INTEGER NOT NULL REFERENCES products(id),

  quantity_ordered INTEGER NOT NULL,
  quantity_received INTEGER NOT NULL DEFAULT 0,

  unit_cost DECIMAL(15, 4) NOT NULL,
  line_total DECIMAL(15, 4) NOT NULL,

  notes TEXT,
  created_at TIMESTAMP NOT NULL DEFAULT NOW(),
  updated_at TIMESTAMP NOT NULL DEFAULT NOW(),

  CONSTRAINT positive_quantity CHECK (quantity_ordered > 0)
);

CREATE INDEX idx_po_items_po ON purchase_order_items(purchase_order_id);
CREATE INDEX idx_po_items_product ON purchase_order_items(product_id);

-- ========================================
-- WAREHOUSE TRANSFERS
-- ========================================
```

```
CREATE TABLE warehouse_transfers (
    id SERIAL PRIMARY KEY,
    organization_id INTEGER NOT NULL REFERENCES organizations(id),
    from_warehouse_id INTEGER NOT NULL REFERENCES warehouses(id),
    to_warehouse_id INTEGER NOT NULL REFERENCES warehouses(id),

    transfer_number VARCHAR(50) NOT NULL,
    status VARCHAR(50) NOT NULL DEFAULT 'pending',
    -- pending, in_transit, completed, cancelled

    initiated_date TIMESTAMP NOT NULL DEFAULT NOW(),
    shipped_date TIMESTAMP,
    received_date TIMESTAMP,

    notes TEXT,

    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW(),
    created_by INTEGER REFERENCES users(id),
    shipped_by INTEGER REFERENCES users(id),
    received_by INTEGER REFERENCES users(id),

    CONSTRAINT different_warehouses CHECK (from_warehouse_id != to_warehouse_id),
    CONSTRAINT unique_transfer_number UNIQUE(organization_id, transfer_number)
);
```

```
CREATE INDEX idx_transfers_org ON warehouse_transfers(organization_id);
CREATE INDEX idx_transfers_from ON warehouse_transfers(from_warehouse_id);
CREATE INDEX idx_transfers_to ON warehouse_transfers(to_warehouse_id);
CREATE INDEX idx_transfers_status ON warehouse_transfers(status);

CREATE TABLE warehouse_transfer_items (
  id SERIAL PRIMARY KEY,
  transfer_id INTEGER NOT NULL REFERENCES warehouse_transfers(id) ON DELETE CASCADE,
  product_id INTEGER NOT NULL REFERENCES products(id),

  quantity_requested INTEGER NOT NULL,
  quantity_shipped INTEGER NOT NULL DEFAULT 0,
  quantity_received INTEGER NOT NULL DEFAULT 0,

  notes TEXT,
  created_at TIMESTAMP NOT NULL DEFAULT NOW(),

  CONSTRAINT positive_quantity CHECK (quantity_requested > 0)
);

CREATE INDEX idx_transfer_items_transfer ON warehouse_transfer_items(transfer_id);
CREATE INDEX idx_transfer_items_product ON warehouse_transfer_items(product_id);
```

2. Missing Requirements - Questions for Product Team

Critical Questions:

- **Multi-tenancy & Data Isolation**

Q: Can users belong to multiple organizations?

Impact: Affects user table structure, permissions model

Current assumption: One user = one organization

Q: Do organizations share any data (e.g., supplier catalogs)?

Impact: Data isolation, security model

Current assumption: Complete data isolation per org

- **Inventory Management**

Q: Do we need batch/lot tracking for products?

Use case: Pharmaceuticals, food items, regulatory compliance

Current: Basic fields added, needs expansion if required

Q: Do we support expiry dates/perishable goods?

Impact: Alerts, FIFO/FEFO logic, waste tracking

Current: Field added in transactions, not in core inventory

Q: What happens to inventory when a product is deleted?

Options: (a) Prevent deletion if inventory exists, (b) Zero out inventory, (c) Soft delete only

Current: Soft delete implemented

Q: Do we need reserve inventory for orders/allocations?

Impact: Separate quantity_reserved column, allocation logic

Current: Quantity_reserved column added

Q: Should we track inventory value/costs (FIFO, LIFO, weighted average)?

Impact: Accounting integration, valuation methods

Current: Unit cost tracked in transactions, no valuation method

- **Product Bundles**

Q: When selling a bundle, how do we handle inventory deduction?

Options: (a) Deduct from components, (b) Track bundle inventory separately

Impact: Inventory calculation complexity

Q: Can bundles contain other bundles (nested)?

Impact: Recursive queries, circular dependency prevention

Current: One level only, constraint prevents self-reference

Q: Can bundle components have variable quantities (e.g., gift boxes)?

Current: Fixed quantities, decimal support added

- **Suppliers**

Q: Can multiple suppliers provide the same product?

Impact: Pricing comparison, preferred supplier logic

Current: Yes, via product_suppliers junction table

Q: Do we need supplier performance tracking (delivery time, quality)?

Impact: Additional metrics tables, reporting

Current: Not implemented

- **Warehouses**

Q: Do warehouses have capacity limits?

Impact: Validation logic, capacity tracking

Current: Not implemented

Q: Do we need location tracking within warehouses?

Impact: More detailed inventory location

Current: Basic bin_location field added

Q: Are there warehouse-specific product restrictions (cold storage)?

Impact: Validation, warehouse capabilities table

Current: Not implemented

- **Pricing & Currency**

Q: Do we support multi-currency pricing per warehouse/region?

Impact: Currency conversion, regional pricing tables

Current: Single currency per product/supplier

Q: Do prices vary by customer/volume?

Impact: Customer pricing tiers, volume discounts

Current: Single sale_price field

- **Access Control**

Q: What's the permission model? (Role-based, warehouse-based, both)?

Examples: Can warehouse staff only see their warehouse? Can viewers only read?

Current: Basic role field, no granular permissions

Q: Do we need approval workflows for inventory adjustments?

Impact: Adjustment requests table, approval chain

Current: Direct adjustment allowed

- **Integrations**

Q: Do we need to integrate with accounting systems (QuickBooks)?

Impact: Export formats, sync tables

Current: Events system supports this

Q: Should we support barcode scanning/mobile apps?

Impact: Barcode fields, API optimization

Current: Barcode fields added

- **Reporting & Analytics**

Q: What reporting periods are needed (real-time, daily, monthly)?

Impact: Aggregation tables, materialized views

Current: Transaction history only

Q: Do we need inventory valuation reports?

Impact: Cost tracking, valuation method

Current: Basic cost fields, no valuation

Q: Should we track inventory turnover rates?

Impact: Analytics tables, calculations

Current: Not implemented

- **Scale & Performance**

Q: What's the expected transaction volume?

Impact: Partitioning strategy, indexing

Current: Indexes added, partitioning commented

Q: How long should we retain transaction history?

Impact: Archival strategy, table partitioning

Current: No archival strategy

- **Business Rules**

Q: Can products have negative inventory (backorders)?

Impact: Constraint removal, backorder tracking

Current: Non-negative constraint enforced

Q: Do we need to track returns/damaged goods separately?

Impact: Return reason tracking, quality control

Current: quantity_damaged field added, no detailed tracking

Q: Should we support serialized inventory (each item tracked individually)?

Impact: Serial number tracking table

Current: Not supported

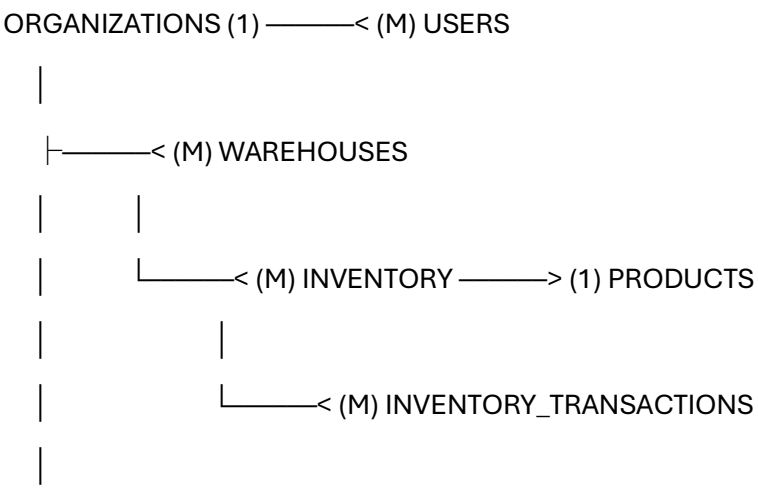
3. Design Decisions & Justifications

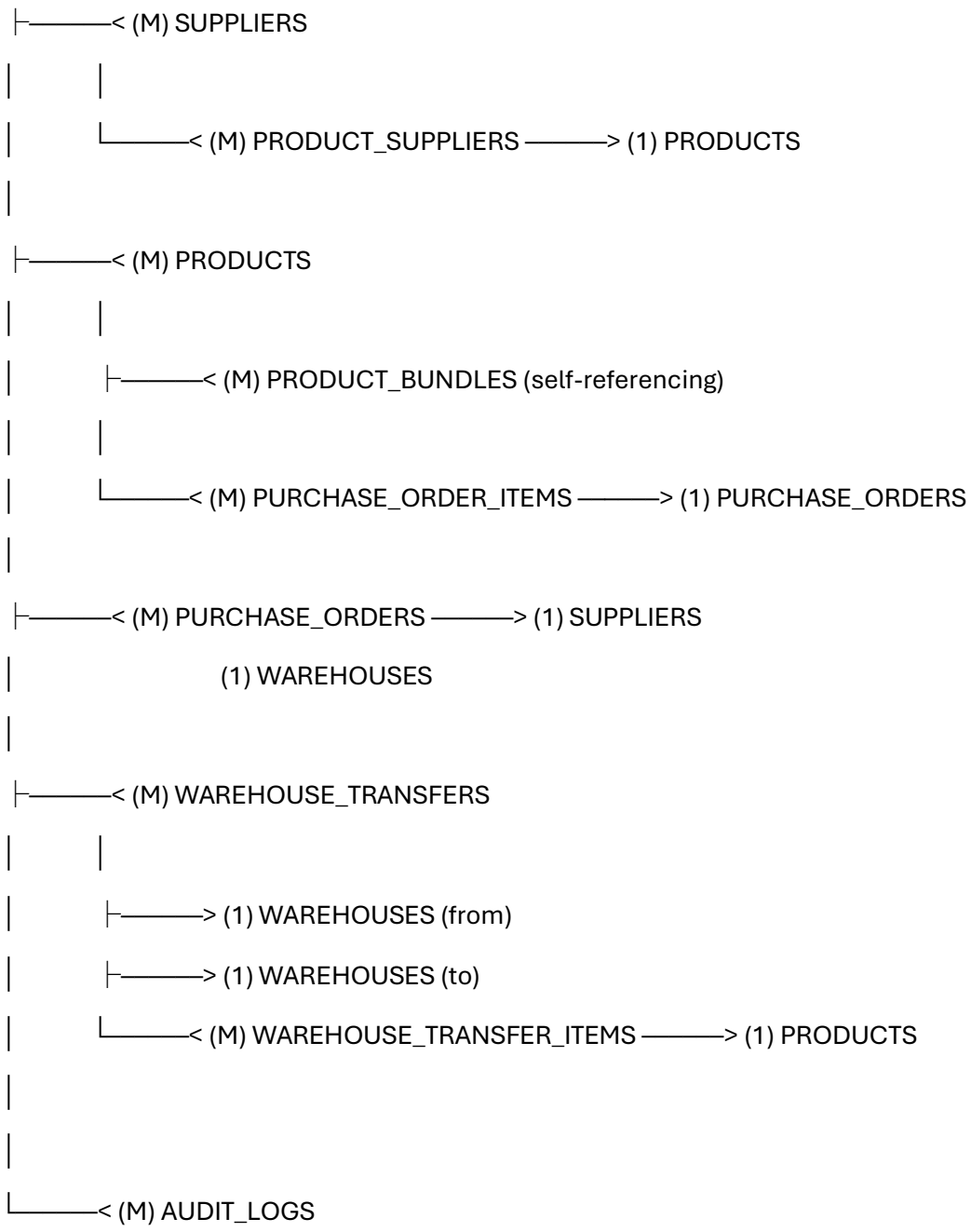
A. Data Types

Choice	Rationale
SERIAL/BIGSERIAL for IDs	Auto-incrementing, easy debugging. BIGSERIAL for high-volume tables (transactions).
DECIMAL(15,4) for money	Exact precision, avoids floating-point errors. 4 decimals for forex rates.

Choice	Rationale
VARCHAR(255) for names	Standard size, indexed efficiently. Shorter for codes/slugs.
TEXT for notes/description	Unlimited length, no arbitrary limits.
JSONB for flexible fields	Schema flexibility, queryable, indexed with GIN.
TIMESTAMP for dates	Timezone-aware, precise to microsecond.
BOOLEAN for flags	Clear intent, indexed efficiently.
INET for IP addresses	Native type, smaller than TEXT, supports network operations.
VARCHAR[] (arrays)	PostgreSQL native, avoids junction tables for simple lists.

B.Entity Relationship Diagram





Part 3: API Implementation - Low Stock Alerts

Implementation (Python/Flask with SQLAlchemy)

```
from flask import Flask, jsonify, request
from sqlalchemy import func, and_, or_, text
from sqlalchemy.orm import joinedload
from datetime import datetime, timedelta
from decimal import Decimal
import logging

app = Flask(__name__)
logger = logging.getLogger(__name__)



# ============================================
# ASSUMPTIONS DOCUMENTED
# ============================================
"""
BUSINESS LOGIC ASSUMPTIONS:
1. "Recent sales activity" = sales in last 30 days
2. "Low stock" = current_stock < threshold for that product
3. Days until stockout = current_stock / avg_daily_sales (if sales > 0)
4. Only alert for active (non-deleted) products
5. Only check warehouses that are active
6. Use preferred supplier if available, otherwise first supplier
7. Consider only quantity_available (not reserved or damaged)

DATABASE ASSUMPTIONS:
1. Using the schema from Part 2 (organizations, products, inventory, etc.)
2. We have a sales_orders table to track recent sales:
```

- sales_orders: id, organization_id, order_date, status

- sales_order_items: order_id, product_id, quantity, warehouse_id

3. Company = Organization (using organization_id)

PERFORMANCE ASSUMPTIONS:

1. This is a read-heavy endpoint (no writes)

2. Results should be cacheable (Redis) for 5-10 minutes

3. Expected to return < 1000 alerts per organization

4. Query should complete in < 2 seconds

"""

```
# ==========================================
# HELPER FUNCTIONS
# ==========================================

def calculate_days_until_stockout(current_stock, avg_daily_sales):
    """
    Calculate estimated days until stock runs out

    Args:
        current_stock: Current available inventory
        avg_daily_sales: Average daily sales over recent period

    Returns:
        int: Days until stockout, or None if can't calculate
    """
```

```
    if avg_daily_sales is None or avg_daily_sales <= 0:
        return None  # Can't predict if no sales history

    if current_stock <= 0:
        return 0  # Already out of stock

    days = current_stock / avg_daily_sales
    return int(days)


def get_preferred_supplier(product_id):
    """
    Get the preferred supplier for a product, or first available supplier

    Args:
        product_id: Product ID

    Returns:
        dict: Supplier information or None
    """
    # Try to get preferred supplier first
    supplier_relation = ProductSupplier.query.filter_by(
        product_id=product_id,
        is_preferred=True
    ).join(Supplier).filter(
        Supplier.is_active == True
    ).first()
```

```
    # If no preferred supplier, get any active supplier
    if not supplier_relation:
        supplier_relation = ProductSupplier.query.filter_by(
            product_id=product_id
        ).join(Supplier).filter(
            Supplier.is_active == True
        ).first()

    if not supplier_relation:
        return None

    supplier = supplier_relation.supplier
    return {
        "id": supplier.id,
        "name": supplier.name,
        "contact_email": supplier.contact_email,
        "contact_phone": supplier.contact_phone,
        "lead_time_days": supplier_relation.lead_time_days,
        "minimum_order_quantity": supplier_relation.minimum_order_quantity,
        "cost_price": float(supplier_relation.cost_price) if supplier_relation.cost_price else None
    }


def get_recent_sales_data(organization_id, days=30):
    """
    Get average daily sales per product per warehouse for recent period
```

```
    Args:
        organization_id: Organization ID
        days: Number of days to look back (default 30)

    Returns:
        dict: {(product_id, warehouse_id): avg_daily_sales}
    """
    cutoff_date = datetime.utcnow() - timedelta(days=days)

    # Query to calculate average daily sales
    # SUM(quantity) / days for each product-warehouse combination
    query = db.session.query(
        SalesOrderItem.product_id,
        SalesOrderItem.warehouse_id,
        func.sum(SalesOrderItem.quantity).label('total_quantity')
    ).join(
        SalesOrder,
        SalesOrderItem.order_id == SalesOrder.id
    ).filter(
        SalesOrder.organization_id == organization_id,
        SalesOrder.order_date >= cutoff_date,
        SalesOrder.status.in_(['completed', 'shipped', 'delivered'])  # Only confirmed sales
    ).group_by(
        SalesOrderItem.product_id,
        SalesOrderItem.warehouse_id
    ).all()
```

```
    # Calculate average daily sales
    sales_data = {}
    for row in query:
        product_id = row.product_id
        warehouse_id = row.warehouse_id
        total_quantity = row.total_quantity
        avg_daily = total_quantity / days
        sales_data[(product_id, warehouse_id)] = avg_daily

    return sales_data


# ============================================
# MAIN ENDPOINT
# ============================================

@app.route('/api/companies/<int:company_id>/alerts/low-stock', methods=['GET'])
def get_low_stock_alerts(company_id):
    """
    Get low stock alerts for a company across all warehouses

    Query Parameters:
        - warehouse_id (optional): Filter by specific warehouse
        - threshold_multiplier (optional): Adjust sensitivity (default 1.0)
        - include_no_sales (optional): Include products with no recent sales (default false)
        - limit (optional): Maximum number of alerts to return (default 100)
```

Returns:

JSON response with low stock alerts and supplier information

"""

```
try:
    # ============================================
    # 1. AUTHENTICATION & AUTHORIZATION
    # ============================================
    # In production, verify JWT token and check user permissions
    current_user = get_current_user()  # Mock function
    if not current_user:
        return jsonify({"error": "Unauthorized"}), 401

    # Verify user has access to this organization
    if not current_user.has_access_to_organization(company_id):
        return jsonify({"error": "Access denied"}), 403

    # ============================================
    # 2. VALIDATE INPUT & PARSE PARAMETERS
    # ============================================
    warehouse_id = request.args.get('warehouse_id', type=int)
    threshold_multiplier = request.args.get('threshold_multiplier', default=1.0, type=float)
    include_no_sales = request.args.get('include_no_sales', default='false').lower() == 'true'
    limit = request.args.get('limit', default=100, type=int)

    # Validate parameters
    if threshold_multiplier <= 0:
```

```
        return jsonify({"error": "threshold_multiplier must be positive"}), 400

    if limit > 1000:
        return jsonify({"error": "limit cannot exceed 1000"}), 400

    # Verify organization exists and is active
    organization = Organization.query.get(company_id)
    if not organization or not organization.is_active:
        return jsonify({"error": "Organization not found"}), 404

    # ==========================================
    # 3. CHECK CACHE (Redis)
    # ==========================================
    cache_key =
f"low_stock_alerts:{company_id}:{warehouse_id}:{threshold_multiplier}:{include_no_sales}"

    # Try to get from cache
    cached_result = redis_client.get(cache_key)
    if cached_result:
        logger.info(f"Cache hit for low stock alerts: {cache_key}")
        return jsonify(json.loads(cached_result)), 200

    # ==========================================
    # 4. GET RECENT SALES DATA
    # ==========================================
    logger.info(f"Calculating recent sales for organization {company_id}")
    sales_data = get_recent_sales_data(company_id, days=30)
```

```
# ==========================================
# 5. BUILD MAIN QUERY FOR LOW STOCK PRODUCTS
# ==========================================

# Base query: Join inventory with products and warehouses
query = db.session.query(
    Product.id.label('product_id'),
    Product.name.label('product_name'),
    Product.sku,
    Product.low_stock_threshold,
    Product.category,
    Inventory.warehouse_id,
    Warehouse.name.label('warehouse_name'),
    Inventory.quantity_available.label('current_stock'),
    Inventory.last_counted_at
).select_from(
    Inventory
).join(
    Product,
    Inventory.product_id == Product.id
).join(
    Warehouse,
    Inventory.warehouse_id == Warehouse.id
).filter(
    # Organization filter
    Product.organization_id == company_id,
```

```
    # Only active products and warehouses
    Product.is_deleted == False,
    Warehouse.is_active == True,

    # Low stock condition: current_stock < (threshold * multiplier)
    Inventory.quantity_available < (Product.low_stock_threshold * threshold_multiplier)
)

# Optional: Filter by specific warehouse
if warehouse_id:
    query = query.filter(Inventory.warehouse_id == warehouse_id)

# Execute query
low_stock_items = query.all()

logger.info(f"Found {len(low_stock_items)} potential low stock items")

# ==========================================
# 6. FILTER BY RECENT SALES ACTIVITY
# ==========================================
alerts = []

for item in low_stock_items:
    product_id = item.product_id
    warehouse_id = item.warehouse_id

    # Get sales data for this product-warehouse combination
```

```
avg_daily_sales = sales_data.get((product_id, warehouse_id), 0)

# Skip products with no recent sales (unless explicitly requested)
if not include_no_sales and avg_daily_sales == 0:
    logger.debug(f"Skipping product {product_id} - no recent sales")
    continue

# ============================================
# 7. CALCULATE DAYS UNTIL STOCKOUT
# ============================================
days_until_stockout = calculate_days_until_stockout(
    item.current_stock,
    avg_daily_sales
)

# ============================================
# 8. GET SUPPLIER INFORMATION
# ============================================
supplier_info = get_preferred_supplier(product_id)

# ============================================
# 9. BUILD ALERT OBJECT
# ============================================
alert = {
    "product_id": item.product_id,
    "product_name": item.product_name,
    "sku": item.sku,
```

```
      "category": item.category,
      "warehouse_id": item.warehouse_id,
      "warehouse_name": item.warehouse_name,
      "current_stock": item.current_stock,
      "threshold": item.low_stock_threshold,
      "days_until_stockout": days_until_stockout,
      "avg_daily_sales": round(avg_daily_sales, 2) if avg_daily_sales > 0 else None,
      "last_counted_at": item.last_counted_at.isoformat() if item.last_counted_at else None,
      "supplier": supplier_info,

      # Additional useful fields
      "urgency": calculate_urgency(days_until_stockout, item.current_stock),
      "recommended_reorder_quantity": calculate_reorder_quantity(
        item.low_stock_threshold,
        item.current_stock,
        supplier_info
      )
    }

    alerts.append(alert)

# ============================================
# 10. SORT BY URGENCY
# ============================================
# Most urgent first: stockout soonest, then lowest stock
alerts.sort(key=lambda x: (
    x['days_until_stockout'] if x['days_until_stockout'] is not None else 999,
```

```
    x['current_stock']
))

# Apply limit
alerts = alerts[:limit]

# ============================================
# 11. BUILD RESPONSE
# ============================================
response = {
    "alerts": alerts,
    "total_alerts": len(alerts),
    "generated_at": datetime.utcnow().isoformat(),
    "parameters": {
        "organization_id": company_id,
        "warehouse_id": warehouse_id,
        "threshold_multiplier": threshold_multiplier,
        "include_no_sales": include_no_sales,
        "sales_period_days": 30
    }
}

# ============================================
# 12. CACHE RESULT
# ============================================
redis_client.setex(
    cache_key,
```

```
            300,  # Cache for 5 minutes
            json.dumps(response)
        )

        # ============================================
        # 13. LOG & RETURN
        # ============================================
        logger.info(f"Returning {len(alerts)} low stock alerts for org {company_id}")

        return jsonify(response), 200

    except Exception as e:
        # ============================================
        # ERROR HANDLING
        # ============================================
        logger.error(f"Error generating low stock alerts: {str(e)}", exc_info=True)

        # Don't expose internal errors to client
        return jsonify({
            "error": "An error occurred while generating alerts",
            "message": "Please try again or contact support",
            "request_id": generate_request_id()  # For debugging
        }), 500


# ============================================
# HELPER FUNCTIONS (continued)
```

```
# ========================================


def calculate_urgency(days_until_stockout, current_stock):
    """
    Calculate urgency level for the alert

    Returns:
        str: "critical", "high", "medium", "low"
    """
    if current_stock <= 0:
        return "critical"

    if days_until_stockout is None:
        return "low"  # No sales data, less urgent

    if days_until_stockout <= 3:
        return "critical"
    elif days_until_stockout <= 7:
        return "high"
    elif days_until_stockout <= 14:
        return "medium"
    else:
        return "low"


def calculate_reorder_quantity(threshold, current_stock, supplier_info):
    """
    Suggest reorder quantity based on threshold and supplier constraints
```

```
    Args:
        threshold: Low stock threshold
        current_stock: Current available stock
        supplier_info: Supplier information including MOQ
    Returns:
        int: Recommended reorder quantity
    """
    # Base calculation: Bring stock back to 2x threshold (safety buffer)
    target_stock = threshold * 2
    needed = target_stock - current_stock

    # Apply supplier minimum order quantity if available
    if supplier_info and supplier_info.get('minimum_order_quantity'):
        moq = supplier_info['minimum_order_quantity']
        if needed < moq:
            needed = moq

    # Round up to nearest 10 for convenience (optional)
    needed = max(10, ((needed + 9) // 10) * 10)

    return needed


def generate_request_id():
    """Generate unique request ID for debugging"""
    import uuid
    return str(uuid.uuid4())
```

- **API Response:**


```
{
  "alerts": [
    {
      "product_id": 123,
      "product_name": "Widget A",
      "sku": "WID-001",
      "category": "Electronics",
      "warehouse_id": 456,
      "warehouse_name": "Main Warehouse",
      "current_stock": 5,
      "threshold": 20,
      "days_until_stockout": 12,
      "avg_daily_sales": 0.42,
      "last_counted_at": "2025-12-15T10:30:00Z",
      "urgency": "medium",
      "recommended_reorder_quantity": 50,
      "supplier": {
        "id": 789,
        "name": "Supplier Corp",
        "contact_email": "orders@supplier.com",
        "contact_phone": "+1-555-0123",
        "lead_time_days": 14,
        "minimum_order_quantity": 50,
        "cost_price": 12.50
```

```
    }
  },
  {
    "product_id": 124,
    "product_name": "Widget B",
    "sku": "WID-002",
    "category": "Hardware",
    "warehouse_id": 456,
    "warehouse_name": "Main Warehouse",
    "current_stock": 0,
    "threshold": 15,
    "days_until_stockout": 0,
    "avg_daily_sales": 2.5,
    "last_counted_at": "2025-12-28T14:20:00Z",
    "urgency": "critical",
    "recommended_reorder_quantity": 50,
    "supplier": null
  }
],
"total_alerts": 2,
"generated_at": "2025-12-29T16:45:30Z",
"parameters": {
  "organization_id": 1,
  "warehouse_id": null,
  "threshold_multiplier": 1.0,
  "include_no_sales": false,
  "sales_period_days": 30
```

```
 }
}
```

 # ==

 # EDGE CASES HANDLED

 # ==

"""

EDGE CASES ADDRESSED:

1. No recent sales data
   - Skip products without sales (unless include_no_sales=true)
   - Return days_until_stockout as null

2. Zero or negative inventory
   - days_until_stockout = 0 (critical urgency)
   - Still include in alerts

3. No supplier assigned
   - supplier field returns null
   - Alert still shown (customer needs to find supplier)

4. Multiple suppliers
   - Use preferred supplier if flagged
   - Otherwise use first available supplier

5. Inactive warehouses/products

- Filtered out in query (is_deleted=false, is_active=true)

6. Very high sales velocity

- Could result in negative days_until_stockout
- Handled by returning 0 for already out of stock

7. Product in multiple warehouses

- Each warehouse gets separate alert
- Allows warehouse-specific action

8. Deleted products with inventory

- Filtered out (is_deleted=false)

9. Concurrent requests

- Cache prevents duplicate calculations
- 5-minute TTL balances freshness vs performance

10. Large result sets

- Limit parameter (default 100, max 1000)
- Pagination could be added if needed

11. Invalid organization_id

- Returns 404 with clear message

12. User without permissions

- Returns 403 after authorization check

13. Database connection failures

- Try-catch with proper error logging
- Returns 500 with generic message (no sensitive info)

14. Cache failures

- Gracefully degrades to direct database query

15. Products with very low thresholds

- threshold_multiplier allows adjusting sensitivity

16. Bundles/component products

- Treated like any other product
- Could extend to check component availability

17. Different time zones

- All timestamps in UTC
- Client responsible for timezone conversion

18. Stale inventory counts

- last_counted_at included in response
- Allows client to show data freshness

19. Supplier with no contact info

- Fields can be null, handled gracefully

20. Circular bundle references

- Not an issue for this endpoint (read-only)

- Prevented by CHECK constraint in DB schema

"""

==========

PERFORMANCE OPTIMIZATIONS

==========

"""

PERFORMANCE CONSIDERATIONS:

1. Caching (Redis)

- 5-minute TTL reduces DB load

- Cache key includes all parameters

2. Efficient queries

- Single query for inventory + products + warehouses

- Separate optimized query for sales data

3. Indexes used

- idx_inventory_low_stock (product_id, warehouse_id, quantity_available)

- idx_products_active (organization_id, is_deleted)

- idx_sales_order_date for recent sales query

4. Pagination

- Limit parameter prevents huge responses

- Default 100, max 1000

5. Lazy loading avoided
   - All needed data fetched in initial queries
   - No N+1 query problems

6. Response size optimization
   - Only essential fields returned
   - Supplier info only if available

7. Future optimizations:
   - Materialized view for low stock products
   - Background job to pre-calculate alerts
   - WebSocket push notifications for critical alerts
   - Database read replicas for heavy read load