Part 1: Identify Issues

1. No SKU Uniqueness Check

Problem: Code inserts a product without checking if the SKU already exists.

Impact: Duplicate SKUs can break downstream logic like lookups, inventory sync, or reporting.

2. Missing Input Validation

Problem: Assumes fields like name, sku, price, warehouse_id, initial_quantity always exist and are valid.

Impact: API will crash with a KeyError or fail silently if invalid types (e.g., string for price) are passed.

3. Price Type Not Handled Properly

Problem: No conversion or validation; if a string is passed, it may silently fail or store incorrect values.

Impact: Price math downstream may break; revenue calculations become unreliable.

4. Products Are Not Meant to Be Bound to a Single Warehouse

Problem: Business logic creates a product tied to one warehouse at creation time.

Impact: Prevents products from being tracked in multiple warehouses later .

5. Inventory Should Be Optional or Separated

Problem: Inventory is tightly coupled with product creation.

Impact: If inventory logic fails, product creation fails entirely—even though product data is valid.

6. No Error Handling

Problem: No try-except or transaction rollback.

Impact: Partial DB writes can lead to inconsistent state (e.g., product created, but inventory not).

7. Double db.session.commit() in the Same Endpoint

Problem: Commits product before creating inventory.

Impact: If inventory commit fails, product remains without any inventory — inconsistent state.

8. Insecure Exposure of Internal product.id

Problem: Directly exposing internal DB IDs without any abstraction.

Impact: Risk of enumeration attacks or misuse unless protected elsewhere.

- **Summary of Fixes**

Issue		Fix
No SKU uniqueness check	→	Check existing SKU before insert
Missing validation	→	Added required field and type checks
Decimal handling	→	Used Decimal() with error catch
Multiple warehouses support	→	Separated product and inventory logic
Atomic transaction	→	Used single commit + rollback
Error handling	→	Added try-except for stability
Clean JSON response	→	Used jsonify + proper HTTP codes

- **Code**

```
@app.route('/api/products', methods=['POST'])
def create_product():
    data = request.get_json()

    # Validate required fields
    required_fields = ['name', 'sku', 'price']
    missing = [field for field in required_fields if field not in data]
    if missing:
        return jsonify({"error": f"Missing fields: {', '.join(missing)}"}), 400

    # Validate price
    try:
        price = Decimal(str(data['price']))
    except (InvalidOperation, ValueError):
        return jsonify({"error": "Invalid price format"}), 400

    # Optional inventory handling
    warehouse_id = data.get('warehouse_id')
    initial_quantity = data.get('initial_quantity', 0)

    try:
```

```
        # Check for existing SKU
        existing = Product.query.filter_by(sku=data['sku']).first()
        if existing:
            return jsonify({"error": "SKU already exists"}), 409

        # Create product
        product = Product(
            name=data['name'],
            sku=data['sku'],
            price=price
        )
        db.session.add(product)
        db.session.flush()  # Get product.id without committing

        # Create inventory record if warehouse info is provided
        if warehouse_id:
            inventory = Inventory(
                product_id=product.id,
                warehouse_id=warehouse_id,
                quantity=initial_quantity
            )
            db.session.add(inventory)

        db.session.commit()
        return jsonify({"message": "Product created", "product_id": product.id}), 201

    except IntegrityError as e:
        db.session.rollback()
        return jsonify({"error": "Database integrity error", "details": str(e)}), 500
    except Exception as e:
        db.session.rollback()
        return jsonify({"error": "Unexpected error", "details": str(e)}), 500
```

Part 2: Schema Design

Table: companies

```
CREATE TABLE companies (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
```

```
);
```

Table: warehouses

```
CREATE TABLE warehouses (
    id SERIAL PRIMARY KEY,
    company_id INTEGER NOT NULL REFERENCES companies(id),
    name TEXT NOT NULL,
    location TEXT,
    UNIQUE(company_id, name) -- Avoid duplicate warehouse names per company
);
```

Table: products

```
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    sku TEXT NOT NULL UNIQUE,
    price DECIMAL(12, 2) NOT NULL,
    is_bundle BOOLEAN DEFAULT FALSE
);
```

Table: inventory

```
CREATE TABLE inventory (
    product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE CASCADE,
    warehouse_id INTEGER NOT NULL REFERENCES warehouses(id) ON DELETE CASCADE,
    quantity INTEGER NOT NULL DEFAULT 0,
    PRIMARY KEY (product_id, warehouse_id)
);
```

Table: inventory_changes

```
CREATE TABLE inventory_changes (
    id SERIAL PRIMARY KEY,
```

```
    product_id INTEGER NOT NULL REFERENCES products(id),
    warehouse_id INTEGER NOT NULL REFERENCES warehouses(id),
    changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    quantity_change INTEGER NOT NULL,
    reason TEXT
);
```

Table: suppliers

```
CREATE TABLE suppliers (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL UNIQUE,
    contact_info TEXT
);
```

Table: product_suppliers (many-to-many between suppliers and products)

```
CREATE TABLE product_suppliers (
    supplier_id INTEGER REFERENCES suppliers(id),
    product_id INTEGER REFERENCES products(id),
    PRIMARY KEY (supplier_id, product_id)
);
```

Table: bundles (for bundle products containing other products)

```
CREATE TABLE bundles (
    bundle_id INTEGER NOT NULL REFERENCES products(id) ON DELETE CASCADE,
    component_product_id INTEGER NOT NULL REFERENCES products(id),
    quantity INTEGER NOT NULL CHECK (quantity > 0),
    PRIMARY KEY (bundle_id, component_product_id),
    CHECK (bundle_id <> component_product_id) -- Prevent self-referencing
);
```