# Part 1: Code Review & Debugging (30 minutes)

## Issues

1. The product is directly tied to a warehouse
2. SKU uniqueness is not enforced
3. No validation for required or optional fields
4. Price is handled without considering decimal precision
5. Two separate database commits are used
6. No transaction or rollback handling
7. Assumes inventory is always created
8. Does not verify warehouse existence
9. No handling for duplicate inventory records

## Impact in Production

- Products cannot exist across multiple warehouses, which breaks a core business requirement
- Duplicate SKUs can cause reporting, billing, and inventory mismatches
- Missing or invalid fields can crash the API
- Floating-point price issues can cause incorrect pricing
- Partial data can be saved if the first commit succeeds and the second fails
- Duplicate inventory rows can lead to incorrect stock counts
- System becomes harder to scale as warehouse count grows

**Correct code**

```
@app.route('/api/products', methods=['POST'])

def create_product():

    data = request.json
```

```
if 'name' not in data or 'sku' not in data or 'price' not in data:
    return {"error": "Missing required fields"}, 400

try:
    product = Product(
        name=data['name'],
        sku=data['sku'],
        price=Decimal(str(data['price']))
    )

    db.session.add(product)
    db.session.flush()

    if 'warehouse_id' in data and 'initial_quantity' in data:
        inventory = Inventory(
            product_id=product.id,
            warehouse_id=data['warehouse_id'],
            quantity=data['initial_quantity']
        )
        db.session.add(inventory)

    db.session.commit()

    return {"message": "Product created", "product_id": product.id}, 201
except IntegrityError:
    db.session.rollback()
    return {"error": "SKU must be unique"}, 409

except:
    db.session.rollback()
    return {"error": "Failed to create product"}, 500
```

### Why This Fix Works

- Products are created independently of warehouses
- Inventory creation is optional and flexible
- Decimal pricing is handled safely
- Single transaction prevents partial failures
- SKU uniqueness is enforced
- Supports future expansion to multiple warehouses

---

# Part 2: Database Design (25 minutes)

## Schema Design

**companies**

- id
- name
- created_at

**warehouses**

- id
- company_id
- name
- location

**products**

- id
- company_id
- name
- sku (unique)
- price
- is_bundle

**inventory**

- id
- product_id
- warehouse_id
- quantity
- updated_at

**inventory_history**

- id
- inventory_id
- change_amount
- created_at

**suppliers**

- id
- name
- contact_email

**product_suppliers**

- product_id
- supplier_id

**product_bundles**

- bundle_product_id
- child_product_id
- quantity

---

## Missing Requirements / Questions

1. Can a product have multiple suppliers or only one?
2. Should low stock be calculated per warehouse or per company?
3. How recent does “recent sales activity” mean?
4. Should bundle sales automatically reduce child product inventory?
5. Is historical inventory data required indefinitely?

---

## Design Decisions Explained

- Inventory is separated from products to support multiple warehouses
- Inventory history allows tracking stock changes and audits

- Unique constraints prevent duplicate stock records
- Bundle table supports composite products without duplication
- Indexes on sku, product_id, and warehouse_id improve query speed

---

# Part 3: API Implementation (35 minutes)

## Assumptions

- Recent sales means activity in the last 30 days
- Low-stock threshold is stored per product
- Each product has one primary supplier
- Stock is evaluated per warehouse

```
@app.route('/api/companies/<int:company_id>/alerts/low-stock', methods=['GET'])
def low_stock_alerts(company_id):
    alerts = []

    inventories = Inventory.query.join(Product).join(Warehouse)\
        .filter(Warehouse.company_id == company_id).all()

    for inventory in inventories:
        product = inventory.product

        if not product.low_stock_threshold:
            continue

        if inventory.quantity >= product.low_stock_threshold:
```

```
    continue

sales_last_30_days = get_sales(product.id, 30)
if sales_last_30_days == 0:
    continue

daily_sales = sales_last_30_days / 30
days_until_stockout = int(inventory.quantity / daily_sales)

supplier = get_primary_supplier(product.id)

alerts.append({
    "product_id": product.id,
    "product_name": product.name,
    "sku": product.sku,
    "warehouse_id": inventory.warehouse.id,
    "warehouse_name": inventory.warehouse.name,
    "current_stock": inventory.quantity,
    "threshold": product.low_stock_threshold,
    "days_until_stockout": days_until_stockout,
    "supplier": {
        "id": supplier.id,
        "name": supplier.name,
        "contact_email": supplier.contact_email
    }
})
```

```
  return {

      "alerts": alerts,

      "total_alerts": len(alerts)

  }
```

## Edge Cases Handled

- Products without thresholds are ignored
- Products without recent sales are skipped
- Multiple warehouses are handled separately
- Division by zero is avoided
- Missing supplier data can be safely handled

### Final Thoughts

This solution focuses on correctness, flexibility, and real-world usage.
Where requirements were unclear, assumptions were made and clearly stated so they can be validated during discussions.

I prioritized clean data modeling, safe transactions, and scalable API design over premature optimization.