Build PO: Technical Documentation

View on Overleaf

Contents

Part I
Introduction

Who This Is For

This guide is for backend engineers, data engineers, and operations team members working on our Purchase Order (PO) generation system. No prior experience with optimization solvers or Pyomo is assumed.

System Overview (Five-Step Walk-Through)

1. A user triggers `GET /products/po/build?supplier=ABC`.
2. The Node backend immediately responds *"PO build running in background"*.
3. The backend fetches Google-Sheet data, builds an input JSON (see examples), and calls the Python solver.
4. The Pyomo solver chooses quantities that satisfy every constraint (MOQ, CPK, supplier min/max) while minimizing the chosen objective.
5. Results are written into the PO spreadsheet: new tab, dashboard formulas, and master sheet row. Errors are logged and/or fallback logic activates if needed.

Glossary

MOQ Minimum Order Quantity — supplier's minimum purchasable units.

CPK Case Pack — units per shipping carton.

PO Purchase Order — finalized request sent to supplier.

Demand Multiple m_i Ratio of (inventory + new PO) to raw demand.

WNPV Weighted Normalized Pairwise Variance — fairness objective that balances listings.

γ_p Binary flag — identifies if product quantity is below threshold (MOQ or Case Pack).

ω_{ij} Binary flag — identifies pairs of listings sharing at least one underordered product.

Developer Tip: New to the codebase? Read the endpoint flow first, then skim the Case 1 example below. Working code locations are called out in `monospace`.

Part II
Automated Build PO Code Architecture

1 Endpoint Flow: GET /products/po/build

Route: `src/api/routes/amazon-products/buildPO.ts`
Controller: `src/AmazonProducts/Controller/BuildPOController.ts`

When a user calls:

`GET {{URL}}/products/po/build?supplier={name}`

the following sequence occurs:

1. **Express Route Handler:** The request is routed to the Express router defined in `buildPO.ts`, which attaches authentication middleware and forwards the request to the `buildPO` controller.
2. **Controller Execution:** The controller (`buildPO`) immediately responds to the client with a 200 OK and a message that the PO build is running in the background. It then asynchronously calls `buildWorkingPO(supplier)`.
3. **PO Build Orchestration:** `buildWorkingPO` (in `buildWorkingPO.ts`) calls `buildAllReadyPOs(supplier)`, which orchestrates the entire PO build process.
4. **Data Preparation:** `buildAllReadyPOs` orchestrates the setup for PO creation by:
   - Fetching all necessary configuration and sheet data concurrently using helper functions and cache. This includes:
     - Spreadsheet IDs and sheet names for the PO Master, Supplier, Destination, Product, and PO Template sheets (via `getSSId`, `getSheetName`, `getSheetRefSheetId`).
     - Column reference objects for each sheet (via `getColumnRefObjIndices`).
     - Row indices for key data (via `getRowIndex`).
   - Collecting all relevant data rows for master, supplier, destination, and product sheets concurrently (via `getRowsConsBySheetId` and the Google API).
   - Fetching the PO dashboard template sheet (via Google API).
   - Using the cache (`Cache.GSConfigData`) to avoid redundant API calls and speed up data access.
   - Determining build mode:
     - If a supplier is specified (manual mode), only that supplier's PO is built.
     - If no supplier is specified (automatic mode), the function loops through all suppliers marked as 'to build' in the PO Builder sheet, preparing tasks for each.
   - Preparing concurrent tasks for each supplier to maximize efficiency and avoid blocking operations (using `Promise.all` and custom delay helpers).
5. **PO Construction:** For each supplier, `buildSinglePOBySupplier` executes a detailed, multi-step process:
   - Fetches warehouse data for all relevant countries and generates a unique PO number concurrently.
   - Copies the PO dashboard template sheet to the main PO spreadsheet and renames it with the new PO number (via Google API).
   - Fetches required products for the supplier by analyzing demand, inventory, and supplier constraints.
   - Builds the final product array for the PO, ensuring all constraints (MOQ, case pack, supplier minimums/maximums) are satisfied.
   - **Optimization:** The core logic for determining the values of the PO is performed by calling the Python solver at
     `src/AmazonProducts/Operations/BuildPO/moqIncreaseV2/solverBranches/WNPV_Solver.py`
     See the "Optimization Model Details" section for a full technical breakdown.
   - The results from the solver are then used to populate the PO sheet with product and supplier details, and order requirements.
   - Adds tracking and invoice links to the PO in the master sheet.
   - Appends the new PO row to an array for later insertion into the master sheet.
   - Handles errors gracefully, logging issues and skipping problematic suppliers if necessary.

6. **Sheet Updates:** After all POs are constructed:
    - The new PO rows are sorted and appended to the master sheet.
    - The PO Builder sheet is updated to remove rows for suppliers whose POs were successfully built.
7. **Logging:** Throughout the process:
    - Progress, key actions, and errors are logged using the `Logger` utility for monitoring and debugging.
    - Each major step (data fetch, PO creation, sheet update) includes log statements for traceability.
    - Errors are caught and reported, ensuring that failures in one supplier's PO do not halt the entire process.

Key Points

- The endpoint is asynchronous: the client receives a response immediately, while the PO build process continues in the background.
- All business logic for PO construction, constraint satisfaction, and sheet updates is handled in the `buildAllReadyPOs` and `buildSinglePOBySupplier` functions.
- Extensive logging and error handling are present throughout the process for traceability.

Part III
Automated Build PO

2 Overview

The `WNPV_Solver` is a Pyomo-based optimization model for determining optimal purchase order (PO) quantities across multiple product listings. It is tailored for warehouse supply chains that must satisfy inventory requirements, supplier constraints, and inter-listing fairness.

About Pyomo: Pyomo is a Python-based open-source modeling language for mathematical optimization. It allows us to define optimization problems using variables, constraints, and objectives in a readable format, then solve them using various optimization solvers (like SCIP or CPLEX).

3 Workflow Summary

1. **Initialization (`__init__`)**: Filters and loads only relevant product data based on listings.
2. **Model Setup (`optimize`)**: Initializes Pyomo variables, constraints, and solver fallback logic.
3. **Constraints**: Case pack equality, per-product MOQ limits, listing demand ratio enforcement, and optional supplier-wide checks.
4. **Objective (`_make_objective`)**: Chooses between three scenarios:
    - Variance-only objective (`variance_one`)
    - WNPV objective with supplier ceiling constraint
    - WNPV objective with supplier floor constraint
5. **Post-processing**: Extracts decision variables, aggregates PO metrics, and logs output.

3.1 Constants and Hyperparameters

- **EPSILON** ($\epsilon = 0.2$): Used in the WNPV objective to assign a small but non-zero weight to $\omega_{ij} = 0$ listing pairs. Prevents hard discontinuities by ensuring even unconstrained pairs still contribute slightly to the objective.
- **VAR_DELTA_MULTIPLIER_OBJECTIVE = 1**: A global scaling factor for the pairwise variance term $(m_i - m_j)^2$ in the WNPV objective. Adjusting this multiplier can increase or reduce the influence of pairwise differences in the overall optimization.
- $\mathbf{M_{gamma}} = 10^6$: A large constant used in gamma (γ_p) constraints to enable conditional logic via the Big M method. It must be chosen based on the expected scale of PO quantities so that it dominates the inequalities when $\gamma_p = 1$.
- $\mathbf{M_{omega}} = 10^6$: A large constant used in omega (ω_{ij}) constraints for pairwise listing logic. It should also reflect the upper bound of Γ_{ij} to ensure logical enforcement is properly triggered.

4 Objective Functions

This solver supports three types of objectives depending on the use case:

4.1 Variance-One Objective (Case 1)

When to use: For straightforward orders with no supplier-level constraints.
Goal: Equalize the adjusted demand multiples m_i across listings as closely as possible.

$$m_i = \frac{\texttt{new\_po\_amount}[i] + \texttt{inventory\_all}[i]}{\texttt{original\_demand}[i]}$$

Objective:

$$Z = \sum_i (m_i - 1)^2$$

This minimizes the squared deviation of each listing's demand multiple from 1. It assumes that all listings should be replenished proportionally to their original demand.

4.2 WNPV Objective (Case 2)

When to use: When the PO must increase to satisfy minimum supplier constraints.
Goal: Minimize inequality in fulfillment across listings by penalizing large differences in demand multiples between pairs of listings, especially when those listings contain products that are constrained (e.g., under MOQ or case pack threshold).

Demand Multiple:

$$m_i = \frac{\texttt{new\_po\_amount}[i] + \texttt{inventory\_all}[i]}{\texttt{original\_demand}[i]}$$

Example: If listing A has a PO of 100 units, 50 units in inventory, and an original demand of 150, then $m_A = \frac{100+50}{150} = 1.0$.

Pairwise Difference:

$$\delta_{ij} = (m_i - m_j)^2$$

Example: If $m_A = 1.0$ and $m_B = 1.4$, then $\delta_{AB} = (1.0 - 1.4)^2 = 0.16$.

Gamma Variables γ_p: γ_p flags whether a product's quantity falls below required thresholds.

γ_p is a binary variable that flags whether a product p is under MOQ or case pack threshold. These are determined by constraints such as:

$$x_p - \texttt{threshold}_p \geq -M_\gamma(1-\gamma_p)$$

$$x_p - \texttt{threshold}_p \leq M_\gamma \cdot \gamma_p - 1$$

If $\gamma_p = 1$, then product p is flagged as underordered.

Example: If `threshold_p` $= 120$ and $x_p = 100$, then $\gamma_p = 1$ since the order is below the threshold.

Definition of $\texttt{threshold}_p$: The threshold is calculated as the stricter of the two fulfillment rules:

$$\texttt{threshold}_p = \max(\texttt{product\_moqs}[p], \texttt{case\_packs}[p])$$

This ensures that a product is only considered sufficiently ordered if it meets both the minimum order quantity and the required case pack size.

Example: If `product_moqs`[X136-35-1101] = 120 and `case_packs`[X136-35-1101] = 12, then:

$$\texttt{threshold}_{\text{X136-35-1101}} = \max(120, 12) = 120$$

The model must order at least 120 units of that product; otherwise, it will be marked underordered by $\gamma_p = 1$.

Omega Variables ω_{ij}: ω_{ij} flags listing pairs that share an underordered product.

ω_{ij} is a binary variable that becomes 1 if any product shared between listings i and j is under the MOQ threshold (i.e., any associated $\gamma_p = 1$). Omega determines whether the squared difference between m_i and m_j is penalized. Constraints ensure:

$$\Gamma_{ij} - N_{ij} \geq -M_\omega(1-\omega_{ij})$$

$$\Gamma_{ij} - N_{ij} \leq M_\omega \cdot \omega_{ij} - 1$$

Where Γ_{ij} is the number of underordered products shared between the two listings, and N_{ij} is the total number of shared products.

Example: If listings A and B both contain 3 shared products, and 2 of them are underordered ($\gamma_p = 1$), then $\Gamma_{AB} = 2$ and $N_{AB} = 3$, resulting in $\omega_{AB} = 1$.

Pairwise Weight:

$$w_{ij} = \begin{cases} \epsilon + (1-\epsilon)\omega_{ij} & \text{if listing } i \text{ or } j \text{ is under MOQ/CPK} \\ 1 & \text{otherwise} \end{cases}$$

Example: If $\omega_{AB} = 1$ and $\epsilon = 0.2$, then $w_{AB} = 0.2 + 0.8 \cdot 1 = 1.0$; If $\omega_{AB} = 0$, then $w_{AB} = 0.2$.

Demand Scaling Factor:

$$d_{ij} = \frac{\texttt{original\_demand}[i] + \texttt{original\_demand}[j]}{2 \cdot \texttt{avg\_original\_demand}}, \quad w_{ij} = w_{ij} \cdot d_{ij}$$

Example: If listing A and B have demands 200 and 100, and the average demand is 150: $d_{AB} = \frac{200+100}{2 \cdot 150} = \frac{300}{300} = 1.0$.

Final Objective:

$$\texttt{WNPV} = \frac{\sum_{i<j} w_{ij} \cdot \delta_{ij}}{\sum_{i<j} w_{ij}}$$

Summary: The WNPV objective encourages fair and proportional replenishment across listings, especially when listings contain constrained products. γ flags underordered products, and ω identifies affected listing pairs, enabling targeted penalization of variance where it matters most.

4.3 Objective Case 3: Decrease Order Amount (Meet Maximum)

When to use: When the PO exceeds the maximum allowable supplier limit and needs a controlled reduction.

Trigger Condition: This objective is used when the original PO total exceeds the supplier's maximum allowed order value:

$$\texttt{original\_dollar} > \texttt{supplier\_max\_order\_amount}$$

Goal: Reduce the PO cost to stay below the supplier's maximum threshold, while still enforcing fairness across listings. We apply the same WNPV structure, but modify the pairwise variance logic to only penalize **overfilled** listings.

Modified Pairwise Difference:

$$\delta_{ij} = \begin{cases} (m_i - m_j)^2 & \text{if } m_i > m_j \\ 0 & \text{otherwise} \end{cases}$$

This asymmetry ensures that only listings with demand multiples higher than their peers are penalized. Listings that are underfilled or equal are not punished.

Example: If $m_A = 1.4$ and $m_B = 1.0$, then:

$$\delta_{AB} = (1.4 - 1.0)^2 = 0.16$$

But if $m_A = 0.8$ and $m_B = 1.0$, then:

$$\delta_{AB} = 0$$

Objective: Same weighted structure as Case 2, but using the above modified δ_{ij}:

$$\texttt{WNPV} = \frac{\sum_{i<j} w_{ij} \cdot \delta_{ij}}{\sum_{i<j} w_{ij}}$$

Constraints:

- A soft floor is added to ensure that the reduced PO still fulfills a reasonable portion of demand:

$$\sum_{l,p} \texttt{price}_p \cdot \texttt{new\_po\_amt}_{l,p} \geq \texttt{po\_cost\_floor}$$

- Initially, $\texttt{po\_cost\_floor} = 0.95 \cdot \texttt{supplier\_max\_order\_amount}$
- On fallback attempts, this floor is gradually lowered to increase feasibility.

Gamma and Omega: The same γ_p and ω_{ij} logic applies as in Case 2:

- $\gamma_p = 1$ if product p is underordered (below MOQ or CPK threshold).
- $\omega_{ij} = 1$ if listings i and j share any underordered product.

Summary: Case 3 modifies WNPV to allow a gentle reduction in total PO size while still enforcing listing-level fairness. It discourages overordering in select listings while preserving proportional distribution and MOQ/CPK compliance.

Constraints:

$$\texttt{new\_po}_p - \text{threshold} \geq -M_\gamma(1 - \gamma_p)$$
$$\texttt{new\_po}_p - \text{threshold} \leq M_\gamma \gamma_p - 1$$

5 Fallback Mechanism

Note: Fallback mode is triggered when the optimization model grows too large or takes too long to solve.

The fallback triggers automatically when optimization complexity (number of variables and constraints) exceeds practical solver limits.

If the number of integer variables exceeds a predefined threshold (e.g., 100), the solver enters a fallback mode to ensure tractable optimization. This mode simplifies the model by disabling several complex components.

If the solver fails or returns infeasible, fallback logic is automatically triggered and errors are logged silently.

Simplifications Applied

- The binary variables γ_p (product under-fulfillment flags) and ω_{ij} (listing-pair activation flags) are disabled.
- All pairwise weights w_{ij} are treated as 1 (uniform weighting), removing conditional logic tied to constraint violations.
- δ_{ij} values are still computed for each listing pair to measure fulfillment variance.
- The solver switches from SCIP to CPLEX for improved performance under large-scale models.

Fallback Objective (NPV with Demand Weighting)

The fallback objective minimizes the average pairwise squared deviation from perfect fulfillment ($m = 1$), while still applying demand-based weighting. The expression is:

$$\texttt{NPV} = \frac{\sum_{i<j} d_{ij} \cdot (m_i - m_j)^2}{\sum_{i<j} d_{ij}}$$

Where:

- $m_i = \frac{\texttt{new\_po\_amount}[i] + \texttt{inventory\_all}[i]}{\texttt{original\_demand}[i]}$ is the demand multiple for listing i.
- $d_{ij} = \frac{\texttt{original\_demand}[i] + \texttt{original\_demand}[j]}{2 \cdot \texttt{avg\_original\_demand}}$ is the demand-based scaling factor for listing pair (i, j).

Note: This fallback objective retains the demand-sensitive fairness of WNPV, but sacrifices selectivity and logic tied to constraint violations.

6 Constraint Inventory

The solver enforces constraints to ensure both satisfaction of business requirements and compliance with the supplier and listing-level rules.

6.1 Case Pack Constraints

This Ensures that total product quantities ordered across all listings are valid multiples of the product's case pack size, as required by suppliers.

Constraint:

$$\sum_{\text{listings } l} x_{l,p} = \texttt{sku\_po\_case\_pack\_multiplier\_vars}[p] \cdot \texttt{case\_packs}[p]$$

Where:

- $x_{l,p}$ is the quantity of product p allocated to listing l.

- `case_packs`[p] is the number of units of product per case pack.
- `sku_po_case_pack_multiplier_vars`[p] is a non-negative integer variable representing how many full cases of p are ordered.

Why this constraint is needed: Suppliers ship only in full cases. If a product appears in multiple listings, the **combined** allocation across those listings must still total a valid case pack multiple.

Example: Product `X136-35-1101` has a case pack size of 12. It appears in both listings `X136-35-A` and `X136-35-B`:

- $x_{\text{X136-35-A,X136-35-1101}} = 9$
- $x_{\text{X136-35-B,X136-35-1101}} = 3$

Then:

$$x_{\text{X136-35-A,X136-35-1101}} + x_{\text{X136-35-B,X136-35-1101}} = 12 = 1 \cdot 12$$

This is valid because 12 is a multiple of the case pack size.

Violation Example: If instead we had:

$$x_{\text{X136-35-A,X136-35-1101}} = 10, \quad x_{\text{X136-35-B,X136-35-1101}} = 3$$

Then the sum is 13, which is not divisible by 12 — invalid order.

6.2 Ratio Constraints (Per-Listing SKU Ratios)

This constraint ensures that the amount of each product allocated to a listing respects the required product proportions — essentially, the "recipe" of the listing.

Why is it needed? Listings are combinations of products (SKUs). Each listing has a specific recipe: a certain quantity of each SKU needed to form one complete listing. For example, a listing might consist of 3 pencils and 4 erasers. If we decide to fulfill this listing with 6 pencils, we must provide exactly 8 erasers to maintain the 3:4 ratio. Otherwise, we are effectively manufacturing a fractional or invalid listing.

What happens if we remove it? Without this constraint, the solver might allocate mismatched quantities — say, 6 pencils and 9 erasers — which can't form whole listings. This leads to wasted units and an inaccurate representation of how listings are fulfilled.

Multi-SKU Listings

$$x_{l,p_i} = \frac{q_{p_i}}{q_{p_1}} \cdot x_{l,p_1}$$

Where:

- x_{l,p_i} = units of product p_i allocated to listing l
- q_{p_i} = required units of product p_i per listing (from recipe)
- p_1 = reference product

Example: For a listing with 2 pencils, 3 pens, and 5 pins:

$$\text{amount of pens} = \frac{3}{2} \cdot \text{amount of pencils}$$
$$\text{amount of pins} = \frac{5}{2} \cdot \text{amount of pencils}$$

Single-SKU Listings

$$x_{l,p} = \texttt{sku\_qty\_mult\_single\_sku\_listing\_var}[l,p] \cdot \texttt{sku\_quantity\_dict\_listing\_product}[l][p]$$

Where:

- $x_{l,p}$ = units of product p allocated to listing l
- $\texttt{sku\_quantity\_dict\_listing\_product}[l][p]$ = required units of p per listing

Example: If a listing requires 3 pencils, valid allocations would be 3, 6, 9, etc.:

$$x_{l,\text{pencil}} = 3 \cdot \text{multiplier}$$

For instance, to produce 2 listings, you'd allocate $3 \times 2 = 6$ pencils. Allocating 4 or 5 pencils would be invalid — you can't produce a fractional listing.

6.3 Product-Level MOQ Constraints

Purpose: Enforce supplier-defined minimum order quantities (MOQs) for each product. These constraints ensure that either a product is not ordered at all, or it is ordered in a quantity greater than or equal to its MOQ.

Constraint (conditional enforcement using binary variable z_p):

$$\begin{aligned} x_p &\geq \texttt{product\_moqs}[p] \cdot (1 - z_p) \\ x_p &\leq \texttt{max\_product\_quantity} \cdot (1 - z_p) \end{aligned}$$

Where:

- $x_p = \sum_{\text{listings } l} x_{l,p}$ is the total amount of product p ordered across all listings.
- $\texttt{product\_moqs}$[p] is the supplier MOQ for product p.
- $\texttt{max\_product\_quantity}$ is a large upper bound.
- $z_p \in \{0, 1\}$ is a binary variable that allows the model to skip ordering a product.

Interpretation:

- If $z_p = 0$: the product is being ordered, and the MOQ must be satisfied.
- If $z_p = 1$: the product is not ordered ($x_p = 0$).

Example: Let product `X136-35-1808` have an MOQ of 24:

- If $z_{\text{X136-35-1808}} = 0$: Then $x_{\text{X136-35-1808}} \geq 24$
- If $z_{\text{X136-35-1808}} = 1$: Then $x_{\text{X136-35-1808}} = 0$

This allows flexibility — the solver can choose not to order the product at all, but if it does, it must meet the MOQ threshold.

6.4 Upper-Bound Constraints on Product Quantity

Purpose: Prevent excessive overordering beyond reasonable limits, while accounting for supplier minimums, original purchase plans, and case pack requirements.

Constraint:

$$\begin{aligned} \text{UB}_1 &= \max(\texttt{MOQ}_p, \texttt{InitialPO}_p) + 10 \\ \text{UB}_2 &= \max(\texttt{MOQ}_p, \texttt{InitialPO}_p) + 5 \cdot \texttt{CPK}_p \\ \text{UB}_p &= \max(\text{UB}_1, \text{UB}_2) \\ x_p &\leq \text{UB}_p \end{aligned}$$

Where:

- x_p is the total PO amount for product p.
- `MOQ_p` is the supplier-defined minimum order quantity.
- `InitialPO_p` is the amount originally planned or expected for product p.
- `CPK_p` is the case pack size for product p.
- `UB_p` is the final calculated upper bound.

Example 1: Let product $p =$ `X136-35-1101`, with the following:

- `MOQ` $= 60$
- `InitialPO` $= 200$
- `CPK` $= 12$

Then:

$$\mathrm{UB}_1 = \max(60, 200) + 10 = 210$$
$$\mathrm{UB}_2 = \max(60, 200) + 5 \cdot 12 = 200 + 60 = 260$$
$$\mathrm{UB} = \max(210, 260) = 260$$

Final constraint:

$$x_{\texttt{X136-35-1101}} \leq 260$$

Example 2: Product $p =$ `X999-00-2020`

- `MOQ` $= 100$
- `InitialPO` $= 70$
- `CPK` $= 8$

$$\mathrm{UB}_1 = \max(100, 70) + 10 = 110$$
$$\mathrm{UB}_2 = \max(100, 70) + 5 \cdot 8 = 100 + 40 = 140$$
$$\mathrm{UB} = \max(110, 140) = 140$$

Final constraint:

$$x_{\texttt{X999-00-2020}} \leq 140$$

6.5 Minimum Inventory Fulfillment

Purpose: Ensure any listing whose on-hand inventory has fallen below its required minimum weeks' supply receives at least one replenishment unit.

Applicability: This constraint is only enforced for listing l if

$$\mathrm{inv\_all}_l < \texttt{min\_inv\_week}[l] \quad \text{and} \quad \mathrm{OriginalPO}_l > 0.$$

Constraint: Let p_1 be the first SKU in listing l. Then

$$x_{l,p_1} \;\geq\; 1.$$

Where:

- x_{l,p_1} is the new PO quantity of SKU p_1 for listing l.
- `min_inv_week`[l] is the minimum inventory weeks threshold for listing l.
- $\texttt{inv\_all}_l$ is the current on-hand inventory for listing l.

- $\texttt{OriginalPO}_l$ is the original PO amount for l.

Example: Listing `X136-35-1101` has

$$\text{inv\_all}_{\text{X136-35-1101}} = 0, \quad \texttt{min\_inv\_week}[\text{X136-35-1101}] = 1, \quad \text{OriginalPO}_{\text{X136-35-1101}} = 200.$$

Since $0 < 1$ and OriginalPO > 0, we enforce

$$x_{\text{X136-35-1101},\,\text{X136-35-1101}} \;\geq\; 1.$$

This guarantees at least one case-pack unit is ordered to restore the minimum inventory buffer.

6.6 Supplier-Level Constraints

Purpose: Enforce global supplier minimums and maximums on total units, case-packs, and spend, ensuring the aggregate PO respects the supplier's requirements.

Constraints:

$$\sum_{\text{listings } l} \sum_{p} x_{l,p} \;\geq\; \texttt{supplier\_moqs}[\text{"supplier\_moq\_units"}]$$

$$\sum_{p} \texttt{sku\_po\_case\_pack\_multiplier\_vars}[p] \;\geq\; \texttt{supplier\_moqs}[\text{"supplier\_moq\_case\_packs"}]$$

$$\sum_{l,p} x_{l,p}\, \texttt{prices}[p] \;\geq\; \texttt{supplier\_moqs}[\text{"supplier\_moq\_price"}]$$

$$\sum_{l,p} x_{l,p}\, \texttt{prices}[p] \;\leq\; \texttt{supplier\_max\_order\_amount} \quad \text{(if a maximum is specified)}$$

Where:

- $\sum_{l,p} x_{l,p}$ is the total units ordered across all listings and products.
- $\sum_{p} \texttt{sku\_po\_case\_pack\_multiplier\_vars}[p]$ is the total number of case-packs ordered.
- $\sum_{l,p} x_{l,p}\, \texttt{prices}[p]$ is the total spend.
- `supplier_moqs` holds the supplier's minimums on units, case-packs, and price.
- `supplier_max_order_amount` is an optional upper bound on spend.

Example: Suppose the supplier requirements are:

$$\begin{aligned} \texttt{supplier\_moq\_units} &= 500, \\ \texttt{supplier\_moq\_case\_packs} &= 10, \\ \texttt{supplier\_moq\_price} &= 1000, \\ \texttt{supplier\_max\_order\_amount} &= 8000, \end{aligned}$$

and our solution allocates:

$$\sum_{l,p} x_{l,p} = 612, \quad \sum_{p} \texttt{sku\_po\_case\_pack\_multiplier\_vars}[p] = 51, \quad \sum_{l,p} x_{l,p}\, \texttt{prices}[p] = 1224.$$

Then:

$$612 \;\geq\; 500, \quad 51 \;\geq\; 10, \quad 1224 \;\geq\; 1000, \quad 1224 \;\leq\; 8000,$$

so all supplier-level constraints are satisfied.

6.7 WNPV-Specific Constraints

Purpose: Introduce auxiliary variables to detect and penalize listings/products that violate MOQ or case-pack thresholds, then measure the pairwise deviation of demand multiples only when such violations occur.

(a) Gamma (γ_p) Identifies whether product p has its total order x_p crossing its threshold.

$$x_p - \texttt{threshold}_p \;\geq\; -M_\gamma\,(1-\gamma_p) \quad \text{and} \quad x_p - \texttt{threshold}_p \;\leq\; M_\gamma\,\gamma_p - 1$$

Where:

- $\texttt{threshold}_p = \begin{cases} \texttt{product\_moqs}[p] + \texttt{case\_packs}[p], & \text{if MOQ} > 0, \\ 2 \cdot \texttt{case\_packs}[p], & \text{otherwise.} \end{cases}$
- M_γ is a large constant.
- $\gamma_p = 1$ if and only if x_p exceeds the threshold.

Example: Suppose $\texttt{threshold}_p = 20$ and $M_\gamma = 1000$:

- If $x_p = 25$, the constraints force $\gamma_p = 1$ (since $25 - 20 > 0$).
- If $x_p = 15$, they allow $\gamma_p = 0$ (since $15 - 20 = -5$ fits the first inequality and fails the second if $\gamma_p = 1$).

(b) Omega (ω_{ij}) Flags whether the total number of "problem" products across listings i and j exceeds the pair size N_{ij}.

$$\Gamma_{ij} - N_{ij} \;\geq\; -M_\omega\,(1-\omega_{ij}) \quad \text{and} \quad \Gamma_{ij} - N_{ij} \;\leq\; M_\omega\,\omega_{ij} - 1$$

Where:

- $\Gamma_{ij} = \sum_{p \in P_{ij}} \gamma_p$ is the count of flagged products in the union of listings i, j.
- $N_{ij} = |P_{ij}|$ is the total number of products across both listings.
- M_ω is a large constant.

Example: Two listings i, j each have one product. If $\gamma_{p_i} = 1$ and $\gamma_{p_j} = 0$, then $\Gamma_{ij} = 1$, $N_{ij} = 2$, and the constraints allow $\omega_{ij} = 0$. Only when $\Gamma_{ij} > 2$ would ω_{ij} be forced to 1.

(c) Delta (δ_{ij}) Measures the squared difference of demand multiples for listings i, j, included only when WNPV is active and at least one listing is flagged.

$$\delta_{ij} = (m_i - m_j)^2$$

Where:

- m_i and m_j are the demand multiples for listings i and j.

Example: If $m_i = 1.2$ and $m_j = 0.8$, then

$$\delta_{ij} = (1.2 - 0.8)^2 = 0.16.$$

7 Output Metrics

The solver logs and returns:

- PO amounts per product/listing
- Total cost, case packs, and units
- Demand multiples and raw demands
- Final objective value and type

8 Optimization Flow

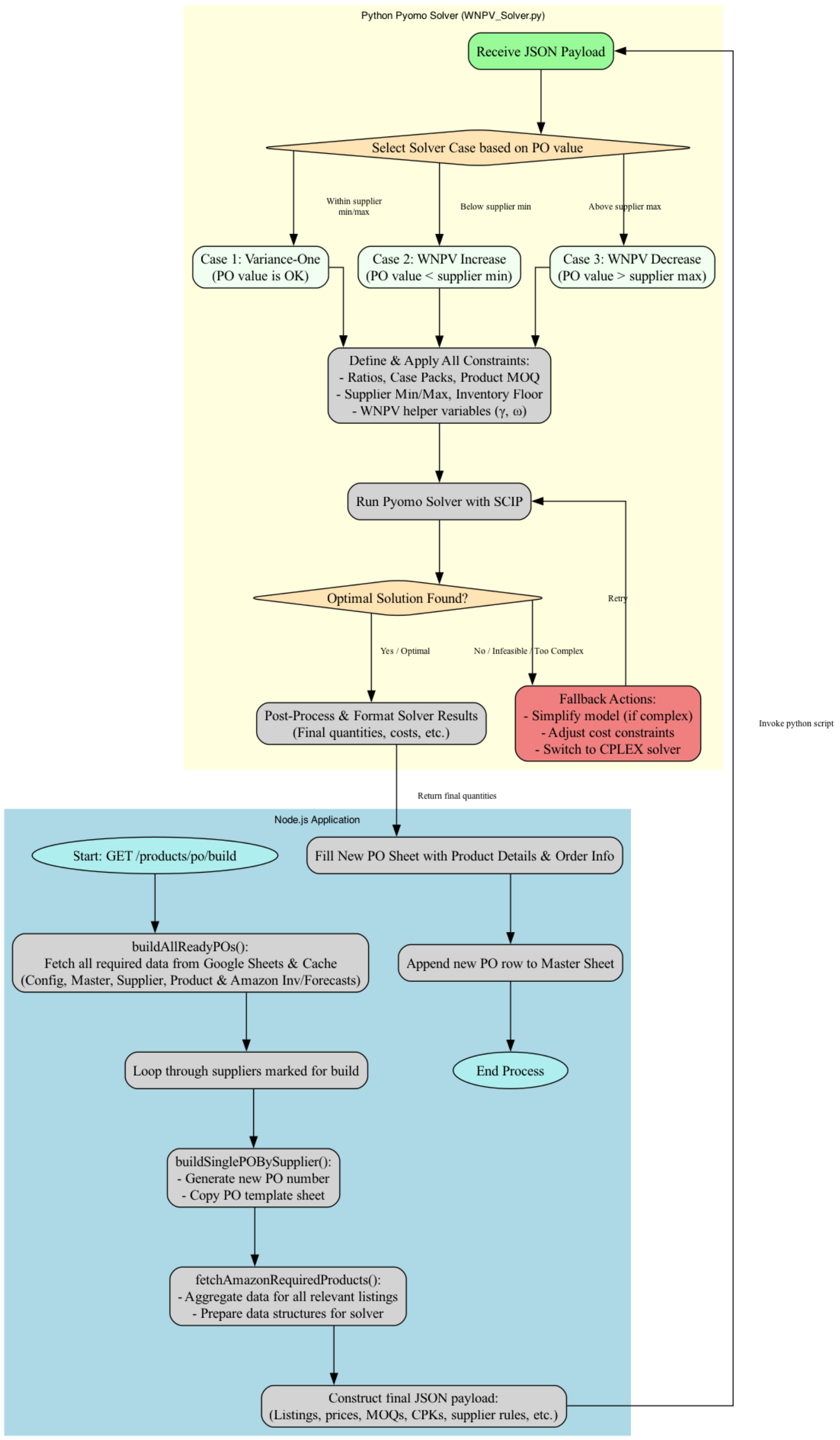
Python Pyomo Solver (WNPV_Solver.py)
Receive JSON Payload
Select Solver Case based on PO value
Within supplier min/max
Below supplier min
Above supplier max
Case 1: Variance-One (PO value is OK)
Case 2: WNPV Increase (PO value < supplier min)
Case 3: WNPV Decrease (PO value > supplier max)
Define & Apply All Constraints: - Ratios, Case Packs, Product MOQ - Supplier Min/Max, Inventory Floor - WNPV helper variables (γ, ω)
Run Pyomo Solver with SCIP
Optimal Solution Found?
Retry
Yes / Optimal
No / Infeasible / Too Complex
Post-Process & Format Solver Results (Final quantities, costs, etc.)
Fallback Actions: - Simplify model (if complex) - Adjust cost constraints - Switch to CPLEX solver
Invoke python script
Return final quantities
Node.js Application
Start: GET /products/po/build
Fill New PO Sheet with Product Details & Order Info
buildAllReadyPOs(): Fetch all required data from Google Sheets & Cache (Config, Master, Supplier, Product & Amazon Inv/Forecasts)
Append new PO row to Master Sheet
Loop through suppliers marked for build
End Process
buildSinglePOBySupplier(): - Generate new PO number - Copy PO template sheet
fetchAmazonRequiredProducts(): - Aggregate data for all relevant listings - Prepare data structures for solver
Construct final JSON payload: (Listings, prices, MOQs, CPKs, supplier rules, etc.)

Figure 1: End-to-end flowchart of the Automated Build PO process, from API request to final sheet update.

Part IV
Examples

Important Testing Note: We include one test case for each objective:

- **Case 1 (Variance-One):** Order remains close to original PO.
- **Case 2 (WNPV - Increase):** Order must increase to meet supplier minimums.
- **Case 3 (WNPV - Decrease):** Order must decrease when listing is overstocked.

No need to revisit known good test cases — only investigate unexpected output. Focus on principles, not preserving specific cases.

9 Example of Case 1 objective

Key insight: This shows simple rounding to nearest case pack multiples when no constraints apply.

Listings

Listing ID	SKU 1	Qty 1	SKU 2-5	Qty 2-5	Min Inv	Inv all	Demand	Orig PO Amt
X136-35-1101	X136-35-1101	1	–	–	1	0	200	200
X136-35-1808	X136-35-1808	1	–	–	1	0	200	200
X136-35-1909	X136-35-1909	1	–	–	1	0	200	200

Products

SKU	MOQ	Case Pack	Unit Price
X136-35-1101	0	12	2
X136-35-1808	0	12	2
X136-35-1909	0	12	2

Supplier MIN: None

Supplier MAX: None

Description

Three listings, each for a single product, all with high demand (200), no MOQ, no supplier constraints. Case Pack of each product is 12. We are expecting to distribute 204 units of product to their respective each listing.

Objective (variance one): 0.0012000000000000023

Explanation of the objective: Use variance-one objective. In this case we have already met the minimum and maximum order amount. We are trying to satisfy the CPK and product MOQ constraints mainly while staying close to the original PO amount.

What the Solver Should Do (Plain English)

No supplier constraints exist, so the only task is to round each 200-unit request *up* to the nearest case-pack multiple. Because the case pack is 12, the smallest feasible quantity 200 is 17 × 12 = 204. Each listing therefore receives 204 units.

Input JSON

```
{"listings":[{"name":"X136-35-1101","products":{"X136-35-1101":1},"po_req_amount":200,"inv_all":0,"demar
            {"name":"X136-35-1808","products":{"X136-35-1808":1},"po_req_amount":200,"inv_all":0,"demanc
            {"name":"X136-35-1909","products":{"X136-35-1909":1},"po_req_amount":200,"inv_all":0,"demanc
 "case_packs":{"X136-35-1101":12,"X136-35-1808":12,"X136-35-1909":12},
 "product_moqs":{"X136-35-1101":0,"X136-35-1808":0,"X136-35-1909":0},
 "prices":{"X136-35-1101":2,"X136-35-1808":2,"X136-35-1909":2},
 "supplier_moqs":{"supplier_moq_price":0,"supplier_moq_units":0,"supplier_moq_case_packs":0},
 "supplier_max_order_amount":{"supplier_max_order_amount":""},
 "do_sup_checks":"Y",
 "min_Inv":4,
 "max_product_quantity":1000000}
```

Variables and Constraints for Case 1 (Variance-One Objective)

Variables

- **new_po_amt_var_listing_product**
  Non-negative integer variables for each listing–product pair; instances:
  - `model.new_po_amt_var_listing_product_X136-35-1101,X136-35-1101`
  - `model.new_po_amt_var_listing_product_X136-35-1808,X136-35-1808`
  - `model.new_po_amt_var_listing_product_X136-35-1909,X136-35-1909`
- **sku_po_case_pack_multiplier_vars**
  Non-negative integer variables enforcing case-pack multiples; instances:
  - `model.sku_po_case_pack_multiplier_vars[X136-35-1101]`
  - `model.sku_po_case_pack_multiplier_vars[X136-35-1808]`
  - `model.sku_po_case_pack_multiplier_vars[X136-35-1909]`
- **sku_qty_mult_single_sku_listing_var**
  Non-negative integer variables scaling original ratios; instances:
  - `model.sku_qty_mult_single_sku_listing_var_X136-35-1101,X136-35-1101`
  - `model.sku_qty_mult_single_sku_listing_var_X136-35-1808,X136-35-1808`
  - `model.sku_qty_mult_single_sku_listing_var_X136-35-1909,X136-35-1909`

Expressions

- **total_units**: total units ordered,

$$\sum_{\ell,p} \text{new\_po\_amt\_var\_listing\_product}_{\ell,p}$$

- **total_cost**: total cost,

$$\sum_{\ell,p} (\text{price}_p \times \text{new\_po\_amt\_var\_listing\_product}_{\ell,p})$$

Constraints

1. **Case-Pack Constraint for X136-35-1101**:
`model.new_po_amt_var_listing_product_X136-35-1101,X136-35-1101`
$=$ `model.sku_po_case_pack_multiplier_vars[X136-35-1101]`
$\times$ case_packs[X136-35-1101]

2. **Case-Pack Constraint for X136-35-1808**:
`model.new_po_amt_var_listing_product_X136-35-1808,X136-35-1808`
$=$ `model.sku_po_case_pack_multiplier_vars[X136-35-1808]`
$\times$ case_packs[X136-35-1808]

3. **Case-Pack Constraint for X136-35-1909**:
`model.new_po_amt_var_listing_product_X136-35-1909,X136-35-1909`
$=$ `model.sku_po_case_pack_multiplier_vars[X136-35-1909]`
$\times$ case_packs[X136-35-1909]

4. **Ratio Constraint for X136-35-1101**:
`model.new_po_amt_var_listing_product_X136-35-1101,X136-35-1101`
$=$ `model.sku_qty_mult_single_sku_listing_var_X136-35-1101,X136-35-1101`
$\times 1$

5. **Ratio Constraint for X136-35-1808**:
`model.new_po_amt_var_listing_product_X136-35-1808,X136-35-1808`
$=$ `model.sku_qty_mult_single_sku_listing_var_X136-35-1808,X136-35-1808`
$\times 1$

6. **Ratio Constraint for X136-35-1909**:
`model.new_po_amt_var_listing_product_X136-35-1909,X136-35-1909`
$=$ `model.sku_qty_mult_single_sku_listing_var_X136-35-1909,X136-35-1909`
$\times 1$

7. **Minimum-Inventory Constraint for X136-35-1101**:
`model.new_po_amt_var_listing_product_X136-35-1101,X136-35-1101`
≥ 1

8. **Minimum-Inventory Constraint for X136-35-1808**:
`model.new_po_amt_var_listing_product_X136-35-1808,X136-35-1808`
≥ 1

9. **Minimum-Inventory Constraint for X136-35-1909**:
`model.new_po_amt_var_listing_product_X136-35-1909,X136-35-1909`
≥ 1

10. **Upper-Bound Constraint for X136-35-1101**:
`model.new_po_amt_var_listing_product_X136-35-1101,X136-35-1101`
≤ 260

11. **Upper-Bound Constraint for X136-35-1808**:
`model.new_po_amt_var_listing_product_X136-35-1808,X136-35-1808`
≤ 260

12. **Upper-Bound Constraint for X136-35-1909**:
`model.new_po_amt_var_listing_product_X136-35-1909,X136-35-1909`
≤ 260

13. **Supplier-Checks Constraints**:

(a) `total_units` ≥ 0

(b) $\sum model.sku\_po\_case\_pack\_multiplier\_vars[p] \geq 0$

(c) `total_cost` ≥ 0

Objective

The Case 1 objective (variance-one) minimized is:

$$Z = \sum_{\ell \in \{\text{X136-35-1101}, \text{X136-35-1808}, \text{X136-35-1909}\}} (d_\ell - 1)^2$$

where the demand multiple d_ℓ for listing ℓ.

10 Example of Case 2 objective

Key insight: Demonstrates increasing order evenly to meet a supplier minimum.

Listings

Listing ID	SKU 1	Qty 1	SKU 2-5	Qty 2-5	Min Inv	Inv all	Demand	Orig PO Amt
X136-35-1101	X136-35-1101	1	–	–	1	0	200	200
X136-35-1808	X136-35-1808	1	–	–	1	0	200	200
X136-35-1909	X136-35-1909	1	–	–	1	0	200	200

Products

SKU	MOQ	Case Pack	Unit Price
X136-35-1101	0	12	2
X136-35-1808	0	12	2
X136-35-1909	0	12	2

Supplier MIN: 10,000

Supplier MAX: None

Description

Three listings, each for a single product, all with high demand (200), and a supplier minimum order amount of 10,000 exceeding the original PO totals. We use the Case 2 (WNPV) objective to increase quantities to satisfy the supplier's minimum order requirement.

Solver Strategy in Plain English

1. Increase every listing by whole-case increments until the total spend meets the $10,000 supplier minimum.
2. Keep all listings scaled equally so their demand-multiples remain identical (fairness enforced by WNPV).

Variables and Constraints for Case 2 (WNPV Upward Objective)

Variables

- **model.new_po_amt_var_listing_product**: Non-negative integer variable for each listing–product pair; instances:
  - `model.new_po_amt_var_listing_product_X136-35-1101,X136-35-1101`
  - `model.new_po_amt_var_listing_product_X136-35-1808,X136-35-1808`
  - `model.new_po_amt_var_listing_product_X136-35-1909,X136-35-1909`
- **model.sku_po_case_pack_multiplier_vars**: Non-negative integer enforcing case-pack multiples; instances:
  - `model.sku_po_case_pack_multiplier_vars[X136-35-1101]`
  - `model.sku_po_case_pack_multiplier_vars[X136-35-1808]`
  - `model.sku_po_case_pack_multiplier_vars[X136-35-1909]`
- **model.sku_qty_mult_single_sku_listing_var**: Non-negative integer scaling original SKU ratios; instances:
  - `model.sku_qty_mult_single_sku_listing_var_X136-35-1101,X136-35-1101`
  - `model.sku_qty_mult_single_sku_listing_var_X136-35-1808,X136-35-1808`
  - `model.sku_qty_mult_single_sku_listing_var_X136-35-1909,X136-35-1909`
- **model.gamma[p]**: Binary indicator for product p below MOQ or case pack; instances: `p` $\in \{X136-35-1101, X136-35-1808, X136-35-1909\}$`.model.omega[l1,l2]` $: Binaryindicatorforlistingpairs; instances: (l1, l2) \in \{(X136-35-1101, X136-35-1808), (X136-35-1101, X136-35-1909), (X136-35-1808, X136-35-1909)\}$.
- `model.delta[l1,l2]: Non-negative real for squared demand differences; same listing pairs as` ω.

Expressions

- **total_units**: $\sum_{\ell,p} \text{new\_po\_amt\_var\_listing\_product}_{\ell,p}$
- **total_cost**: $\sum_{\ell,p} (\text{price}_p \times \text{new\_po\_amt\_var\_listing\_product}_{\ell,p})$
- **total_diff**: $\sum_{i<j} w_{ij}\, \delta[l_i, l_j]$
- **total_weight**: $\sum_{i<j} w_{ij}$
- **wnpv**: $\dfrac{\text{total\_diff}}{\text{total\_weight}}$

Constraints

1. **Case-Pack Constraints**: For each product p:

$$\sum_{\ell} \text{new\_po\_amt\_var\_listing\_product}_{\ell,p} = \text{model.sku\_po\_case\_pack\_multiplier\_vars}[p] \times \text{case\_packs}_p.$$

2. **Ratio Constraints**: For each single-SKU listing (ℓ, p):

$$\text{new\_po\_amt\_var\_listing\_product}_{\ell,p} = \text{model.sku\_qty\_mult\_single\_sku\_listing\_var}_{(\ell,p)} \times \text{original SKU qty}.$$

3. **Minimum-Inventory Constraints**: If $\text{inv\_all}_\ell < \text{min\_inv}$:
$$\text{new\_po\_amt\_var\_listing\_product}_{\ell,p_1} \geq 1.$$
4. **Upper-Bound Constraints**: For each product p:
$$\sum_\ell \text{new\_po\_amt\_var\_listing\_product}_{\ell,p} \leq \max(\text{moq}_p,\, \text{init}_p) + \max(10, 5 \times \text{case\_packs}_p).$$
5. **Supplier-Checks Constraints**:
   - total_units $\geq$ supplier_moq_units
   - $\sum_p$ sku_po_case_pack_multiplier_vars$[p]$ $\geq$ supplier_moq_case_packs
   - total_cost $\geq$ supplier_moq_price
   - (if max order) total_cost $\leq$ supplier_max_order_amount
6. **Gamma Constraints**: For each product p:
$$\text{new\_po}_p - \text{threshold}_p \geq -M_\gamma(1-\gamma[p]),$$
$$\text{new\_po}_p - \text{threshold}_p \leq M_\gamma \gamma[p] - 1.$$
7. **Omega Constraints**: For each listing pair (l_i, l_j):
$$\sum_{p \in P_{ij}} \gamma[p] - N_{ij} \geq -M_\omega(1-\omega[l_i,l_j]),$$
$$\sum_{p \in P_{ij}} \gamma[p] - N_{ij} \leq M_\omega \omega[l_i,l_j] - 1.$$
8. **Delta Constraints**: For each listing pair (l_i, l_j):
$$\delta[l_i,l_j] = (d_{l_i} - d_{l_j})^2.$$
9. **McCormick Constraints**: Linearize $z = \omega\delta$ if used.

Objective

Minimize the weighted normalized pairwise variance:
$$Z = \frac{\sum_{i<j} w_{ij}(d_{l_i} - d_{l_j})^2}{\sum_{i<j} w_{ij}},$$
where
$$w_{ij} = \omega[l_i,l_j] + \epsilon(1-\omega[l_i,l_j]).$$

11 Example of Case 3 objective

Key insight: Shows controlled decrease of order amount to respect a supplier's maximum constraint.

Listings

Products

SKU	MOQ	Case Pack	Unit Price
X136-35-1101	0	12	2
X136-35-1808	0	12	2
X136-35-1909	0	12	2

Listing ID	SKU 1	Qty 1	SKU 2–5	Qty 2–5	Min Inv	Inv all	Demand	Orig PO Amt
X136-35-1101	X136-35-1101	1	–	–	1	0	200	200
X136-35-1808	X136-35-1808	1	–	–	1	0	200	200
X136-35-1909	X136-35-1909	1	–	–	1	0	200	200

Supplier MIN: None

Supplier MAX: 500

Description

Three listings, each for a single product, all with high demand (200), and a supplier maximum order amount of 500 below the original PO totals (600). We use the Case 3 (WNPV downward adjustment) objective to decrease quantities to satisfy the supplier's maximum order requirement.

Solver Strategy in Plain English

1. Remove whole cases *evenly* from each listing until the spend drops below the $500 supplier cap.
2. Penalize only the listings that would otherwise stay over-filled—this is handled by the modified WNPV objective.

Variables and Constraints for Case 3 (WNPV Downward Adjustment)

Variables

- **model.new_po_amt_var_listing_product**: Non-negative integer variable for each listing–product pair.
  Instances: `(X136-35-1101, X136-35-1101)`, `(X136-35-1808, X136-35-1808)`, `(X136-35-1909, X136-35-1909)`.
- **model.sku_po_case_pack_multiplier_vars**: Non-negative integer variable representing case-pack multiples for each product.
  Instances: `X136-35-1101`, `X136-35-1808`, `X136-35-1909`.
- **model.sku_qty_mult_single_sku_listing_var**: Non-negative integer variable for the single-SKU listing ratio.
  Instances: `X136-35-1101`, `X136-35-1808`, `X136-35-1909`.
- **model.gamma**[p]: Binary variable indicating whether product p is underordered (i.e., fails MOQ or CPK threshold).
  Instances: `X136-35-1101`, `X136-35-1808`, `X136-35-1909`.
- **model.omega**[l_i, l_j]: Binary variable indicating whether any product shared between listings l_i and l_j is underordered.
  Instances: `(X136-35-1101, X136-35-1808)`, `(X136-35-1101, X136-35-1909)`, `(X136-35-1808, X136-35-1909)`.
- **model.delta**[l_i, l_j]: Non-negative real variable representing the squared difference in demand multiples between listing l_i and l_j.
  Instances: Same pairs as `omega`.

Expressions

- **total_diff**: Weighted sum of squared demand differences:
  $\sum_{i<j} w_{ij}\,\delta[l_i, l_j]$.
- **total_weight**: Sum of weights:
  $\sum_{i<j} w_{ij}$.
- **wnpv**: Weighted normalized pairwise variance:
  $\frac{\text{total\_diff}}{\text{total\_weight}}$.

Constraints

1. **Case-Pack Constraints** (each product p):

$$\sum_{\ell} \text{new\_po\_amt\_var\_listing\_product}_{\ell,p} = \text{model.sku\_po\_case\_pack\_multiplier\_vars}[p] \times \text{case\_packs}_p$$

2. **Ratio Constraints** (single-SKU listing (ℓ, p)):

$$\text{new\_po\_amt\_var\_listing\_product}_{\ell,p} = \text{model.sku\_qty\_mult\_single\_sku\_listing\_var}_{(\ell,p)} \times \text{original SKU qty}$$

3. **Minimum-Inventory Constraints** (if $\text{inv\_all}_\ell < \text{min\_inv}$):

$$\text{new\_po\_amt\_var\_listing\_product}_{\ell,p_1} \geq 1$$

4. **Upper-Bound Constraints** (each product p):

$$\sum_{\ell} \text{new\_po\_amt\_var\_listing\_product}_{\ell,p} \leq \max(\text{moq}_p,\ \text{init}_p) + \max(10, 5 \times \text{case\_packs}_p)$$

5. **Supplier-Checks Constraints**:
   - $\text{total\_units} \geq \text{supplier\_moq\_units}$
   - $\sum_p \text{sku\_po\_case\_pack\_multiplier\_vars}[p] \geq \text{supplier\_moq\_case\_packs}$
   - $\text{total\_cost} \geq \text{supplier\_moq\_price}$
   - $\text{total\_cost} \leq \text{supplier\_max\_order\_amount}$

6. **Gamma Constraints** (each product p):

$$\begin{aligned} \text{new\_po}_p - \text{threshold}_p &\geq -M_\gamma(1 - \gamma[p]), \\ \text{new\_po}_p - \text{threshold}_p &\leq M_\gamma \gamma[p] - 1 \end{aligned}$$

7. **Omega Constraints** (each listing pair (l_i, l_j)):

$$\begin{aligned} \sum_{p \in P_{ij}} \gamma[p] - N_{ij} &\geq -M_\omega(1 - \omega[l_i, l_j]), \\ \sum_{p \in P_{ij}} \gamma[p] - N_{ij} &\leq M_\omega \omega[l_i, l_j] - 1 \end{aligned}$$

8. **Delta Constraints** (each listing pair (l_i, l_j)):

$$\delta[l_i, l_j] = (d_{l_i} - d_{l_j})^2$$

9. **McCormick Constraints**: Linearize $z = \omega\delta$ if introduced.

Objective

Minimize the weighted normalized pairwise variance (downward adjustment):

$$Z = \frac{\sum_{i<j} w_{ij}(d_{l_i} - d_{l_j})^2}{\sum_{i<j} w_{ij}},$$

where

$$w_{ij} = \omega[l_i, l_j] + \epsilon(1 - \omega[l_i, l_j]).$$

Debugging Tips

- **Blank PO sheet?** Template copy failed in `buildSinglePOBySupplier`.
- **Quantities way off?** Inspect the JSON handed to `WNPV_Solver.py`.
- **Solver very slow?** Fallback may have triggered—look for ”Entering fallback mode” in logs.
- **Supplier skipped?** Verify filters in `getRowsConsBySheetId`.

Common Edits and Where To Make Them

- **Include / exclude listings** → `buildAllReadyPOs`.
- **Tweak optimization objective** → `WNPV_Solver.py` *(method _make_objective)*.
- **Change fallback thresholds** → `optimize()` inside the solver.

Part V
Feature Development History (Jira Summary)

The current Build PO solver is the result of two major development initiatives, tracked in Jira as GSD-919 and GSD-920. These epics addressed different facets of the PO generation problem, leading to the robust, multi-case system described in this document.

12 GSD-919: Handling Supplier Minimums and Maximums

This story tackled the core optimization challenge: how to intelligently adjust a purchase order when its initial value falls outside the supplier's financial constraints.

Primary Goal To create a system that could either increase a PO's total value to meet a supplier's minimum requirement or decrease it to stay under a maximum, while ensuring the changes were distributed ”fairly” across the listings.

Key Developments & Challenges

- **Evolution of the Objective Function:** The initial approach of using a single objective for both increasing and decreasing the PO proved too complex and yielded suboptimal results. The team went through several iterations:
    1. A two-objective model, which was difficult to balance.
    2. Simpler variance calculations (from mean and from one).

3. The final, successful approach: **Weighted Normalized Pairwise Variance (WNPV)**. This was chosen for its ability to penalize "unfairness" between listings, especially when some listings contained products that were driving constraint violations (e.g., being under MOQ).

- **Solver Limitations and Fallback Strategy:** A significant challenge arose when implementing the WNPV objective. The use of binary variables (γ and ω) in the objective function created a non-convex problem that the standard CPLEX solver could not handle due to its degree limitations. This led to:
  - An experimental phase with the **SCIP solver**, which could handle higher-degree objectives but proved to be too slow for complex, real-world POs.
  - The development of a **hybrid fallback system**. The final strategy, as documented, is to first attempt a solution with the full, complex WNPV model. If the problem is too large (i.e., has too many integer variables) or the solver fails, the system falls back to a simpler model (disabling γ and ω) that can be solved quickly by CPLEX.
- **Guiding Constraints:** To help the solver converge, the concepts of a **cost ceiling** (when increasing the PO) and a **cost floor** (when decreasing the PO) were introduced. These constraints narrow the search space and are designed to be relaxed iteratively if a solution is not immediately found.

13 GSD-920: Core Refactoring and the "Within Bounds" Case

Running in parallel with GSD-919, this story focused on improving the overall architecture, performance, and correctness of the "standard" case where the PO value is already within the supplier's min/max limits.

Primary Goal To refactor the legacy codebase into a more robust, maintainable, and faster system, and to define a specific, efficient objective for the common "within-bounds" scenario.

Key Developments & Challenges

- **Architectural Refactoring:** The codebase was significantly cleaned up and organized into smaller, single-responsibility functions. This improved readability and made the complex logic easier to manage and debug.
- **API Reliability:** The development process uncovered intermittent 502 errors from Google Sheets API calls, likely due to rate-limiting. A robust, global retry mechanism with exponential backoff delays was implemented to handle these transient errors, making the entire data-fetching process more reliable.
- **Specific Objective for Case 1:** For the "within-bounds" case, a simpler and more direct objective was defined: minimize the sum of squared deviations from a perfect demand multiple of 1 ($\sum(m_i - 1)^2$). This "variance-one" objective effectively tells the solver to make the smallest possible changes to the original PO while satisfying product-level MOQ and Case Pack constraints.
- **Specific Upper-Bound Constraint:** A unique and detailed upper-bound constraint on product quantity was developed specifically for this case to prevent the solver from making excessive, unnecessary additions to any single product.

Synthesis and Final Outcome

The work from both tickets converged into the unified, three-case solver that is the subject of this documentation. The system first checks the PO's total value against the supplier's min/max constraints to decide which objective function to apply (WNPV for increase/decrease, Variance-One for within bounds). The architectural improvements and retry logic from GSD-920 provide a stable foundation, while the advanced optimization and fallback strategies from GSD-919 handle the more complex edge cases.