ChatGPT

ABI RGM Pricing Optimizer – Deep Research & Build Plan

Scope & Success Criteria

The goal is to build a **SKU-level price optimization engine** for AB InBev's portfolio in the target market (Colombia) that **maximizes total Margin of Contribution (MACO)** while respecting all business constraints. This extends the Round 1 elasticity analysis into an actionable pricing tool, reallocating a user-defined overall Price Increase (PINC) across SKUs optimally [1]. The scope includes **all SKUs in the market (both ABI and competitors)**, using their own- and cross-price elasticities to predict volume shifts for any price changes. Key success criteria are: **(1)** Achieving a **higher total MACO** than the current baseline (i.e. improved profit) [2]; **(2)** Keeping volume and market share changes within allowed bounds (to avoid volume loss or excessive gains that violate capacity/market assumptions) [3]; **(3)** Maintaining established **price architecture hierarchies** (brand segment ladder and pack size ladder) to ensure a sensible portfolio pricing structure [4]. In summary, success means an implementable price plan that **boosts profitability** (MACO up) under realistic market reactions, **meets all hard constraints**, and **preserves brand equity signals**, thereby providing revenue managers a data-driven pricing recommendation ready for integration into the BrewVision tool.

Inputs & Assumptions

Data Inputs: We will leverage all available unmasked data files to calibrate and run the optimizer: - **Elasticity Matrix (CSV)** – This contains the own-price and cross-price elasticities for each SKU (target) with respect to other SKUs' prices, as derived in Round 1. It covers ~153 ABI SKUs' own elasticities (negative) and their cross-elasticities with both ABI and competitor SKUs (positive for substitutes) [5] [6]. This is the core input for demand sensitivity. - **Sell-out Data (Train & Test)** – Transactional sales data (volume in hectoliters, value, etc.) by SKU, used to establish the baseline **current year volume** for each SKU. We will use the latest available period (likely full-year 2024) as the **base volume** for optimization. Base volumes for ABI and competitors are computed from this dataset (e.g. total ABI volume in 2024) to determine the "current year" baseline and market share. - **Price List (Excel)** – Contains each SKU's **current price to consumer (PTC)**, retailer markup, pack size, etc. We will extract each SKU's current PTC (e.g. Aguila 330mL NRB = 2000 COP) and its pack details. The retailer markup (e.g. ~21.8% for that SKU) will be used to compute manufacturer's net revenue per unit【13†】. We assume the listed **markup, discount%, and excise%** are representative and remain constant for the scenario (approx. 8% average discount and 19.9% excise as a fraction of GTO, based on base year data aggregation). These values, along with a fixed **VAT of 19%** [7], will feed into the net revenue calculations. - **Sell-in Data (Excel)** – Provides financial metrics per SKU (Gross Turnover, Discount, Excise, Net Revenue, Variable Industrial Cost (VIC), Variable Logistics Cost (VLC), etc.) for past years. From this, we extract **base MACO** (NR – (VIC+VLC)) for the current year and **base VILC** (Variable Industrial & Logistics Cost). For example, using 2024 data we calculate total base-year MACO. We will assume these **unit costs increase by 3.78%** going forward (the provided VILC growth rate) [8]. The base VILC per HL for each SKU (or on average) will be scaled by +3.78% to compute the "target" cost per volume in the optimization. - **IHS Macroeconomic Data** – Contains industry and economic indicators (e.g. price indices). We specifically use the **industry volume elasticity factor** provided: a **0.56** coefficient linking

price change to industry volume change [9]. This informs the industry volume constraint (see below). No other macro data is directly used in the model except for this industry elasticity assumption.

User Inputs: The tool will allow two main user inputs through the front-end: - **National Price Increase (PINC)** – a target overall portfolio price increase (0–6% range) [10]. This is the percentage uplift in weighted average price that the revenue manager wants to achieve. The optimizer will distribute this increase across SKUs. We assume the user will specify a value (e.g. 4%) within the allowed 0–6% range, and that this is an *overall* increase for ABI's portfolio. (If the chosen PINC is infeasible with other constraints – e.g. 6% might conflict with volume limits – the model will handle it as discussed later.) - **Price Change Bounds** – optional bounds on individual SKU price changes. By default, we will enforce a reasonable range, e.g. **minimum = current PTC – 300, maximum = current PTC + 500** (in local currency), as suggested [11]. These bounds ensure no single SKU's price moves beyond feasible limits (e.g. no more than a 300 COP price cut or 500 COP increase by default). The UI will allow adjusting these bounds globally or for specific SKUs (e.g. if a manager wants to freeze certain SKUs' price, they can set that SKU's min = max = current price). We assume any user-specified bounds will still allow some solution space; if extremely tight bounds are set on all SKUs (leaving no room to reallocate PINC), the optimizer might not find a feasible solution (we will warn or require at least some flexibility).

Key Assumptions: - **Baseline "Current Year":** We treat the latest actual year (2024) as the base scenario for volumes, revenue, MACO, and share. All "base" values (volume, NR, MACO, etc.) are computed from 2024 data. The optimization's "new" scenario will be compared to these for constraint evaluation (e.g. volume change ±% vs 2024). - **Competitor Prices Static:** We assume competitor brands do **not change their prices** during our optimization. The competitive response is only via volume: if ABI raises prices, some volume may shift to competitors (captured by cross-elasticities), but competitor pricing itself remains constant. This means our decision variables only cover ABI SKU prices; competitor volumes are endogenous outcomes (not controlled directly) determined by ABI's moves. - **Linearity of Elasticity:** We assume the elasticity matrix provides a locally linear approximation of demand response. For the small-to-moderate price changes we consider (0–6% overall, typically within ±10% per SKU), a linear model is acceptable. Thus, we will use a **linear demand model** where each SKU's % volume change is a linear function of % price changes of all SKUs, weighted by the elasticities [9]. (We acknowledge that true demand might be slightly non-linear, but this linear approximation greatly simplifies the optimization and is reasonable for modest price shifts.) - **Net Revenue Calculation:** We will use the provided formula for net revenue per unit: `NR/unit = ((PTC / (1+markup)) * (1 – Discount% – Excise%)) / (1+VAT)` [7]. We assume **markup, discount%, and excise% remain at their base values** for the new scenario (i.e. trade margins and tax rates are stable). This means any PTC change propagates proportionally to manufacturer's net revenue. For example, if a SKU's current net revenue is 50% of PTC (after removing retailer margin, discounts, excise, VAT), we apply that same ~50% factor to the new PTC to get new NR/unit. - **Cost and MACO:** "VILC" (variable industrial and logistics cost) is assumed to increase by **+3.78%** as given [8]. We will apply this uplift uniformly to all SKUs' unit costs. In other words, each SKU's cost per HL in the new scenario = base cost per HL * 1.0378 (regardless of volume change). We assume no economies or diseconomies of scale in unit cost at these volume change levels – cost per unit is treated as fixed (with the inflationary increase) and does not depend on volume. This allows us to compute MACO for any scenario as `MACO = NR – VILC` straightforwardly [12]. - **Hierarchy Definitions:** We adopt the given definitions for segment and size categories. Each SKU has a **segment** (Value, Core, Core+, Premium, Super Premium, etc.) from the `Category` field in the price list. We assume "Value", "Core", "Core+", "Premium", "Super Premium" are the relevant segments in the beer portfolio hierarchy, and we will enforce ordering among these five (other categories like non-alcoholic, malt, RTD, etc., which don't fit in this ladder, will not be constrained against the beer segments). Each SKU is also classified by **pack size**: **Small** if <300 mL, **Regular** if (for cans) 300–399 mL or (for bottles "RB/NRB") 300–599 mL, **Large** if ≥400 mL for cans or ≥600 mL for bottles [13]. These definitions will be applied to each SKU based on its `capacity_number` and

`package_type`. We assume this classification covers all SKUs (e.g. kegs or very large formats would fall under Large). - **Exclusions:** The optimization assumes stable external conditions – no new competitors, no supply constraints, and no major marketing/promo changes during the period. We focus solely on base price changes. Dynamic effects (e.g. how quickly volume responds, long-term brand power) are out of scope for this pricing optimizer (addressed in the separate LTE task). We also assume all price changes we recommend can be implemented without regulatory or logistical issues and that the price elasticity relationships remain applicable in the new price regime.

Optimization Model

Decision Variables: For each ABI SKU *i*, we introduce a decision variable for its new price. To naturally enforce the 50-unit price step rule, we define the decision variable as an integer change increment: let **ΔP_i** = number of 50-COP steps by which we change SKU *i*'s consumer price. Thus, **Price_new_i = Price_base_i + 50 * ΔP_i**. Here ΔP_i can be positive (price increase), negative (price decrease) or zero, and it will be bounded by some range (e.g. –6 ≤ ΔP_i ≤ +10 if base±{300,500}). Using an integer variable inherently ensures **price changes in multiples of 50** currency units [14]. This formulation turns our pricing problem into an **MILP (Mixed Integer Linear Program)** once we linearize the objective and constraints. We choose absolute price change (in currency) as the variable (instead of % change) because the 50-unit step is more naturally enforced in absolute terms.

Objective Function: Maximize the **total MACO** for ABI. We express total MACO as the sum across all ABI SKUs of `(Net Revenue – Variable Cost)` for the new scenario. Expanding this: - *Net Revenue for SKU i* = **NR_per_unit_i(new)** × **Sales_Units_i(new)**. We compute NR_per_unit(new) using the formula with the new PTC: `NR/unit_i(new) = ((Price_new_i/(1+markup_i)) * (1 – Discount%_i – Excise%_i)) / (1+VAT)` [7]. Most terms are constants from the input data (markup and tax rates, etc.), so effectively *NR/unit_i* is roughly a linear function of Price_new_i. *Sales_Units(new)* = *Volume_HL(new)* × (100000 / pack_size_ml) [15] (since 1 HL = 100,000 mL) – this converts the volume in hectoliters to number of units sold. - *Variable Cost for SKU i* = **Cost_per_HL_i(new)** × **Volume_HL_i(new)**. We set Cost_per_HL(new) = Cost_per_HL(base) × 1.0378 (adding 3.78% inflation) [8]. We derive Cost_per_HL(base) for each SKU from Sell-in data (VIC+VLC divided by volume). If granular SKU-level cost is unavailable, we will use an average cost per HL for each segment or the total ABI average, scaled by SKU volume for total cost – since we are primarily concerned with total MACO, a uniform cost per HL across SKUs will still yield correct relative improvements.

The objective **Maximize Σ_{i∈ABI} [NR_i(new) – Cost_i(new)]** is inherently **non-linear**, because *NR_i(new)* depends on Price_new_i and Volume_new_i, and *Volume_new_i* in turn depends on all SKUs' price changes (via cross-elasticities). To handle this, we linearize the demand response and use the elasticity matrix. We assume a linear demand model of the form:

$$\%\Delta Volume_i \approx \sum_{j \in \text{all SKUs}} E_{i,j} \times \%\Delta Price_j,$$

where *E_{i,j}* is the elasticity of SKU i's volume with respect to SKU j's price. This is equivalent to:

$$Volume_i(new) = Volume_i(base) \times \Big(1 + E_{i,i}\frac{\Delta P_i}{Price_i(base)} + \sum_{j \neq i} E_{i,j}\frac{\Delta P_j}{Price_j(base)}\Big).$$

We will use this linear equation to compute each SKU's new volume (both ABI and competitor SKUs) as **linear functions of the ΔP variables**. The elasticity values E come directly from the Round 1 output (with typical own-price E_{i,i} around –1 to –5 for ABI SKUs [16], and various cross-elasticities indicating cannibalization between SKUs). By plugging these volume expressions into the revenue and cost calculations, we can transform the objective into a form suitable for an MILP or MIQP solver.

Linearization Approach: To keep the model linear, we will initially treat the volume–price relationship as linear (as above) and the net revenue per unit as *approximately linear* in price. Notably, if discount% and excise% are constants, then NR/unit is essentially a constant fraction of the consumer price. For example, if after markup and taxes, AB InBev nets ~50% of PTC, then raising PTC by 1 currency unit raises NR/unit by ~0.5 units. We will use the base values to determine this relationship for each SKU. This means **NR_i(new)** can be linearized as c_i * Price_new_i * Volume_new_i(base proportion) plus minor adjustments. The tricky part is the product of Price and Volume (bilinear term). To solve this, we have two options: 1. **Mixed Integer Quadratic Programming (MIQP):** Formulate it as a quadratic objective (price * volume) with linear constraints. Modern solvers like Gurobi can handle MIQP efficiently for our problem size. If available, we would leverage such a solver to directly maximize the quadratic profit function. 2. **Approximation via Piecewise Linearization:** If using a purely linear MILP solver (e.g. CBC via PuLP or OR-Tools CP-SAT which requires linear objectives), we can approximate the profit contribution in a piecewise linear manner. One approach is to pre-compute the **marginal profit impact** of each possible price step for each SKU (considering elasticity effects) and introduce additional variables to select those steps. However, given the complexity and time constraints, a more straightforward approach is preferable.

Our plan is to attempt using an **open-source solver** that can handle quadratic objectives (for example, the Python OR-Tools CP-SAT solver can handle linear constraints with a linear objective; we may try using its CP-SAT for MILP and verify if an MIQP extension is feasible, or use Pyomo with an IPOPT solver for a nonlinear solve). If that proves difficult, we will linearize by evaluating the objective gradient at the base and assume small changes (essentially treating the objective as linear in price over the limited range). In practice, because cost is fixed per unit, maximizing MACO is very close to maximizing total net revenue (since cost just shifts the objective by a constant times volume). And maximizing net revenue given a fixed total volume change can be linearized (it tends to allocate price where elasticity * revenue trade-off is best). Thus, we anticipate that **the optimal solution will raise prices more on less elastic (inelastic) SKUs and possibly lower on highly elastic SKUs to gain volume**, which aligns with intuition.

Given these simplifications, we will **formulate the optimization as an MILP**: - Decision vars: ΔP_i (integers). - Constraints: linear (see next section). - Objective: initially maximize a linear approximation of ΔMACO. We will verify the solution by computing the true MACO and, if needed, do a brief local search around the MILP solution to adjust any minor differences (since the search space is discrete but small around optimum). This hybrid approach ensures we find a near-optimal solution that truly maximizes MACO.

Finally, note that the **baseline MACO** (for current prices) is a known constant from data. We will compute it beforehand. The optimization effectively focuses on maximizing the *incremental MACO* above base (subject to being ≥ 0). We will include a constraint to ensure MACO_new $\geq$ MACO_base [2] (so the solver doesn't choose a trivial or negative outcome even if, say, PINC=0).

Constraints & Business Rules

We incorporate **all hard constraints** from the problem statement as strict requirements in the model, and we also respect the **soft constraints** by adding appropriate constraints to maintain the intended hierarchy (these can be slightly relaxed if absolutely necessary, but we aim to satisfy them fully). Below are the constraints and how we implement each:

Hard Constraints: (All must be satisfied by the optimizer solution) - **Industry Volume Tether:** The total **industry volume** (ABI + competitors) in the new scenario should not decline by more than 1% from the base. We connect the solution to the given industry model: `New_Industry_Volume = (1 – 0.56 * ΔP_avg) * Old_Industry_Volume` [9], where ΔP_avg is the weighted average price change in the market. In practical terms, we will enforce **New_Total_Volume ≥ 0.99 × Old_Total_Volume**. Our model inherently computes new volumes for each SKU (including competitors via cross-elasticities), so we will sum all volumes and constrain that sum to ≥ 99% of the base sum. This guarantees the industry-wide volume drop ≤1% [17]. The 0.56 factor is essentially built into our elasticity-based volume changes (it's consistent with an average elasticity for the whole market), but we use the ≤1% drop as a direct constraint for clarity. - **Financial Target (MACO):** The **total MACO in the optimized scenario must exceed the base year MACO** [2]. We will calculate base MACO from the data (e.g. sum of 2024 NR – VILC for ABI) and include a constraint: **Total_MACO_new ≥ Total_MACO_base + ε**. (We can allow a tiny ε margin or 0, effectively "≥"). This ensures the solution delivers financial improvement. In practice, the solver is maximizing MACO, so it will naturally aim well above this threshold; this constraint just rules out any corner cases where a solution that barely changes prices (or lowers them) could be considered if PINC was zero. Essentially, profit must not go down. - **ABI Volume Target:** The **total ABI sales volume** should not *decrease by more than 1%* nor *increase by more than 5%* vs base [18]. We will enforce: **0.99 × ABI_Volume_base ≤ ABI_Volume_new ≤ 1.05 × ABI_Volume_base**. This keeps ABI's volume within –1% to +5% of current levels. The rationale is to avoid severe volume loss (which could hurt distribution and share) or unrealistically large volume gains (which may be infeasible in production capacity or market demand). This constraint will interact with the PINC: raising prices too much tends to shrink volume, so the 5% upper bound likely won't bind unless the optimizer tried a price cut strategy (which is unlikely given we target an overall increase; but if the user input was a small PINC or negative – though negative PINC isn't allowed by input). - **Total Price Increase (PINC) Allocation:** We ensure the solution's overall weighted price increase matches the user-defined PINC (within the 0–6% range) [10]. We will implement this by a weighted average price constraint. Specifically, we compute the **volume-weighted average price** of ABI's portfolio before and after. The user input PINC = X% means: Σ(Price_new_i * base_weight_i) = (1+X%) * Σ(Price_base_i * base_weight_i). Here base_weight_i can be each SKU's share of base revenue or volume – since PINC is defined at a portfolio level, we'll use revenue weights (so that it truly reflects an average price to consumer). A simpler implementation is: **Σ(P_new_i * Volume_base_i) = (1+PINC) * Σ(P_base_i * Volume_base_i)**, which ensures the weighted price across constant base volume is up by PINC%. This is slightly different from a true post-optimization weighted price (since volumes shift), but it's a close proxy and keeps the constraint linear. Essentially, it forces the distribution of price changes to use up the "budget" of PINC. If needed, we can adjust weights iteratively (but since PINC is small, base volume weights are fine). The user input is constrained 0% ≤ PINC ≤ 6% [19], and we will validate that input. The model will treat it as an equality (or very tight range) so that we fully utilize the allowed increase. (If a slight deviation is needed to meet other hard constraints, we might allow leaving some PINC unallocated, but by default we try to hit it exactly.) - **Market Share Preservation:** ABI's **market share should not drop by more than 0.5 percentage points** [20]. We calculate ABI share = ABI_volume / Total_industry_volume. Base share comes from data (for 2024, ABI's volume share ~92.4%, for example). We enforce **Share_new ≥ Share_base – 0.005**. This is a non-linear constraint (because share_new = ABI_vol_new / total_vol_new), but we can linearize it: impose **ABI_vol_new ≥ (Share_base – 0.005) * Total_vol_new**. Expanding using known Share_base

constant (e.g. 0.924), it becomes ABI_vol_new – 0.9195 * Total_vol_new ≥ 0 (which is linear). This constraint prevents losing significant share to competitors. Intuitively, if ABI raises prices too much and competitor volumes surge, this will trigger the share constraint and the solver will pull back some price increases on SKUs that are causing large share loss. The 0.5% point cap ensures ABI remains at least ~**99.5%** of its original share. - **Pricing Multiples:** As noted, **all price changes must be in increments of 50** currency units [14]. This is guaranteed by our decision variable formulation (ΔP_i is integer). We will also explicitly restrict ΔP_i to multiples that achieve that (i.e. ΔP_i ∈ ℤ). The solver inherently handles this because ΔP_i is an integer variable. No fractional price changes will occur. - **SKU Price Bounds:** Each SKU's new price will be constrained within a lower and upper bound based on current price and user setting. By default, we include **Price_new_i ≥ Price_base_i – 300** and **≤ Price_base_i + 500** (if using the example bounds) [11]. These translate to bounds on ΔP_i (e.g. ΔP_i ≥ –6 and ≤ +10 for a base price where 50*6 =300, 50*10=500). If the user provides custom bounds (different values or specific per SKU adjustments), those are plugged in accordingly. These constraints ensure no extreme price cuts or hikes beyond what management is willing to consider, keeping recommendations realistic.

Soft Constraints (Brand Hierarchy): To maintain a logical price structure, we add constraints that enforce the **Net Revenue per Hectoliter (NR/HL) ladders**: - **Segment Ladder:** We ensure that the **NR/HL of each segment** follows the order **Value < Core < Core+ < Premium < Super Premium** [21]. In practice, NR/HL is proportional to price per liter (since discount/excise are percentages and VAT is fixed). So this essentially means **price per liter (or per HL) for any Premium SKU should exceed that of any Core+ SKU, which exceeds any Core SKU, etc.** We will implement this by imposing constraints between segments. For example, let PPL_i = (Price_new_i) / (pack_size_i in liters). For every pair of segments, we enforce a margin: e.g. for Core segment vs Value segment, **min(PPL_core) ≥ max(PPL_value) + δ**. To simplify, we can take representative SKUs or average price per liter in each segment. One approach: introduce auxiliary variables for each segment's average NR/HL, and constrain those: NR/HL_seg(Core) > NR/HL_seg(Value), etc. Another approach is pairwise for each SKU: for every Value SKU and every Core SKU, enforce PPL_core > PPL_value. That could be a lot of constraints, but since segments are few, we might pick the **highest-priced Value SKU** and **lowest-priced Core SKU** and enforce a separation. We will likely use the latter approach: identify edge cases (the most expensive Value and cheapest Core, etc.) based on base prices or iteratively, then ensure ordering. The spirit is to guarantee no overlap. We will incorporate a small buffer δ (could be 0 or a tiny positive value) to avoid equality – typically Premium should be distinctly above Core+. These constraints reflect the "established hierarchy" that higher segments carry higher unit revenues. Our solution will respect this unless absolutely impossible (which it shouldn't be, as we have freedom to adjust within PINC budget). - **Package Size Ladder:** Ensure **Small packs > Regular > Large** in terms of NR/HL [22]. This means, for example, a 250 mL can (small) should have a higher per-liter price than a 330 mL bottle (regular), which in turn is higher than a 750 mL bottle (large), reflecting the typical volume discount for larger packages. We will classify each SKU as Small, Regular, or Large using the given rules [13] and then impose similar constraints: all Regular SKUs' PPL should be ≤ all Small SKUs' PPL, and all Large SKUs' PPL ≤ all Regular SKUs' PPL. Again, we might implement this by ensuring the **minimum** PPL of small > maximum PPL of regular, etc. In case there are edge anomalies (e.g. a very high-priced large pack vs a low-priced small pack of different segments), the segment hierarchy and size hierarchy combined will straighten these out. We treat these as *soft constraints*, but in implementation we will likely add them as linear constraints (which the solver must satisfy). If the optimization finds it impossible to maximize profit without violating a ladder (unlikely given some slack in pricing), we will favor satisfying the ladder because maintaining hierarchy is strategically important. Thus, effectively, we handle them like additional hard constraints in the model (with the understanding that they are "soft" from a business perspective and could be relaxed if needed).

By including these constraints, the optimizer's recommended **price architecture will inherently satisfy business rules**. For example, if the algorithm tried to give a bigger % increase to a Value brand

than a Premium brand (because maybe the Premium brand is very elastic), the ladder constraint will cap that, forcing perhaps a different allocation or leaving some PINC unused for Value. We will verify after solving that indeed **Value segment NR/HL < Core < Core+ < Premium < Super Premium** and **Small pack NR/HL > Regular > Large**, as required [22].

Finally, we will ensure the solution respects **all** of the above. The MILP solver will only return solutions that meet all hard constraints. The soft constraints are also encoded, so the returned solution should naturally maintain the hierarchy. We will double-check computed outputs (volume changes, share, etc.) to confirm the constraints (for instance, confirm volume change is within ±1%, share drop $\leq$0.5%, etc., as a validation step after solve). If any soft constraint looks marginal (e.g. two segments nearly equal in NR/HL), we might post-adjust by a minimal price tweak (which should be possible without breaking hard constraints) to create a clear separation.

Solver Strategy

We plan to use a **mathematical optimization approach** (operations research solver) to guarantee finding the optimal prices under the above model. Given the problem structure, this is a **Mixed Integer Programming** problem (integer due to price steps, potentially quadratic due to price*volume in objective). Our strategy is as follows:

Solver Choice: We will implement the optimization in Python using a library such as **PuLP (with CBC solver)** or **OR-Tools CP-SAT**. Both handle MILP problems and are open-source. OR-Tools' CP-SAT is particularly powerful for integer problems and should handle ~150 integer variables and a few thousand linear constraints efficiently on a laptop. If the objective remains quadratic, we might leverage **Pyomo** with an open-source quadratic solver or attempt using **Gurobi** (if a license is available for the hackathon) for MIQP. However, to keep things simple and "laptop-friendly," we intend to linearize as discussed and use CP-SAT which guarantees an optimal solution to the MILP. CP-SAT also allows adding logical constraints easily if needed (though our constraints are mostly linear inequalities).

Computational Feasibility: The number of decision variables is on the order of the number of ABI SKUs (~153). Each SKU's volume equation connects to all others' price changes, which means potentially ~23k elasticity constraint terms (153×152). Many cross-elasticities are small or zero (in practice each SKU has significant elasticity with only a subset of others, like within brand or segment), so we can prune negligible terms below a threshold to reduce model density. But even 20k linear terms is trivial for modern solvers. We estimate the MILP will have a few hundred constraints (from volume relations, plus one for each hard constraint and bounds). This is well within the capability to solve in seconds. We expect the optimization to solve nearly instantly (< 10 seconds) on a standard laptop with CBC or CP-SAT, given the problem size. We will test performance with the full model; if needed, we can further simplify (e.g. grouping some SKUs or fixing decisions for very low-volume SKUs that don't impact objectives much). But we anticipate it's not necessary.

Solving Workflow: We will build the model using OR-Tools' Python interface or PuLP, which allows us to define variables, constraints, and objective in a readable way. Key steps: 1. **Data Loading & Precomputation:** Read the elasticity matrix into a structure for quick lookup of E_{i,j}. Prepare base volumes, base share, base MACO from data. Pre-calculate constants like NR/unit factors (c_i for each SKU) and cost per HL. 2. **Variable Definition:** Create an integer variable ΔP_i for each ABI SKU with bounds as derived from price limits. 3. **Volume Constraints:** For each SKU *i* (including competitor SKUs if we simulate them), add a linear equation for Volume_new_i as a function of all ΔP. For ABI *i*, we can directly compute Volume_new_i and also use it in objective; for competitor *k*, we'll have ΔP_k = 0 (since they don't change price) but their volume will still change based on others' ΔP. We can either include

competitor volumes explicitly with their own (non-decision) variables linked by elasticity constraints or compute competitor volume on the fly when evaluating share – we may include them as continuous variables for completeness so that share constraint becomes linear in those variables. 4. **Objective Construction:** Add the objective to maximize sum_i (NR_i(new) – cost_i(new)). In a linearized form, this might be Σ_i [(NR/unit base * ΔP_i * something) + …] – but more straightforwardly, we might instruct the solver to maximize an expression for MACO_new. If using CP-SAT, we can linearize piecewise or simply iterate over possible ΔP combinations if needed (though brute force is not feasible for 153 SKUs). Instead, we'll rely on the solver's branch-and-bound. If the objective is linearized, it's directly maximized. If we go with CP-SAT, which actually maximizes integer linear objective, we need that linear approximation – we will likely use the base volume * price * c_i approach for that. We will test a small scenario to ensure the linear approximation doesn't mislead; given constraints on volume and share, the solution likely won't involve extreme nonlinear effects. 5. **Add Constraints:** All the constraints listed (PINC, volume bounds, share, etc.) will be added to the model. For soft constraints, add them as normal constraints (ensuring feasibility space is restricted to hierarchy-valid solutions). 6. **Solve:** Call the solver. If using OR-Tools CP-SAT, we'll set an optimality focus (it finds proven optimal or very close solutions quickly). We'll also set a reasonable time limit (if needed, e.g. 30 seconds) though we expect it to finish much faster. 7. **Solution Extraction:** Retrieve the optimal ΔP for each SKU and compute the resulting new prices. Then calculate all outcome metrics (volumes, NR, MACO, etc.) from the solution to present to the user.

Ensuring Optimality: By using an MILP approach, we ensure a **global optimum** (the solver's guarantee). There's no risk of getting stuck in a local optimum as might happen with heuristic or gradient approaches. We will double-check the returned solution's MACO by plugging it into the full non-linear calculation. If there is any discrepancy due to linearization, we will evaluate if adjusting any ΔP by ±1 could improve actual MACO. Because the search space is relatively small per SKU and the solver's answer will be near optimal, a quick "fine-tune" check can catch any off-by-one-step issues. If, for instance, the MILP gave a solution that is 0.1% suboptimal in actual MACO, we can manually or via a short program test neighboring price combinations to see if MACO improves (this is feasible given the small delta and constraints). We expect this won't be necessary with a well-chosen linear proxy, but it's a contingency plan.

Feasibility & Slack: In case the model as formulated is **infeasible** (no solution satisfies all constraints), it likely means the user inputs are conflicting (e.g. PINC too high). The solver will report infeasibility. Our strategy then is to relax the least critical constraint: for example, we could treat the volume ±1% as a strict requirement and if unsolvable, allow volume to drop a bit more (since the industry tether formula actually would allow ~3% drop if PINC=6%). Because the hard constraints come from business, we would first try to relax the PINC (the user target) slightly. We can implement a fallback where if no solution, we reduce the effective PINC target until feasibility (essentially not using the full increase). Alternatively, we could allow the volume constraint to extend to –3% (since industry model with 6% price increase suggests ~3.3% drop) as a "soft" overflow with a penalty in objective. In this plan, to keep things simple, we will **validate user input** before solving: e.g., if user enters 6% PINC, we know from the industry elasticity that this likely violates the 1% volume drop constraint. We will warn the user or automatically adjust the volume constraint to –3% soft in that case. We prefer not to surprise the user by ignoring their PINC, so likely we'll include that check and suggest using a lower PINC or understanding that the solution might hit the volume bound. Our solver can also incorporate a slight slack for volume drop and try to minimize it if necessary (using a big-M and penalty). Given these considerations, we expect to handle feasibility robustly.

Integrality & Rounding: The solver provides integer ΔP values, so new prices will automatically be multiples of 50. No further rounding needed – the output prices are directly implementable. All other calculated metrics (volume, share) will be floating-point results. We might round final volumes to, say, 1

hectoliter precision or so when displaying (for neatness), but internally we keep full precision in calculations.

In summary, our solver strategy uses a **deterministic, exact optimization method** to explore the price reallocation possibilities. This ensures the **PINC is optimally utilized to maximize profit** given the demand response. By contrast, a heuristic approach (like raising price by elasticity * margin logic) might miss interactions – our MILP will consider cross-effects globally. The approach is data-driven and guarantees that if there is a way to increase MACO under the rules, the solver will find the best one.

UI/UX Plan

We propose a **dual-path front-end approach** to cater to both rapid prototyping and a polished final product: 1. **Streamlit Web App (MVP):** For the initial implementation and internal testing, we will develop a simple Streamlit application in Python. This provides a quick, interactive UI that runs in the same environment as the optimization code. The UI will have a clean, minimal design: - **Inputs:** A slider or numeric input for the desired PINC (0 to 6%). Fields to adjust global price bounds (min and max change) and possibly a multi-select dropdown to choose specific SKUs to constrain (for advanced use, e.g. freeze price of a certain strategic SKU). Defaults will be pre-filled (e.g. PINC 3%, bounds –300/+500). - **Trigger:** A prominent "Run Optimization" button. On click, the app will execute the optimization pipeline with the chosen inputs. - **Outputs:** Once solved, the app will display results in two forms: - **Summary KPIs:** At the top, a concise summary of key outcome metrics: e.g. **Volume Change**, **Net Revenue Change**, **MACO Change** (absolute and % vs base), and **Market Share Change**. This can be shown as a small table or bullet points, highlighting that all are within constraints (perhaps color-coded green if within allowed range or red if not, though our solution will ensure they are within range by design). - **Detailed Results Table:** A scrollable table listing each SKU (or each brand/SKU combination) with columns for Old Price, New Price, Price Change (%), Old Volume, New Volume, Volume % change, NR/HL old & new, MACO/HL old & new, etc. We will include the required fields: old vs new price, and the impact on Volume, NR, NR/HL, MACO, MACO/HL [23] . This table allows the user to see granular effects. We will make it sortable and filterable (Streamlit can integrate interactive DataFrames or we can use Ag-Grid component) so that, for example, the user can sort by largest price increase or largest volume loss to see which SKUs were most affected. - **Visualization – SKU Price Architecture:** Below the table, we will present a chart visualizing the "before and after" price architecture across SKUs [24] . One effective visualization is a **bar chart** of price per liter for each SKU, grouped by segment. For instance, we can have the x-axis list SKUs (or a representative SKU per segment/pack) sorted by segment and price, and y-axis as price (per liter). We will use different colors for old vs new price bars for each SKU to show the change. This will clearly illustrate that, e.g., all Premium SKUs moved to a higher price per liter than all Core SKUs, etc., and how much each moved. We might also overlay markers for NR/HL if needed, but since NR/HL is proportional to consumer price by a factor, the visualization of price suffices to show the architecture. If too many SKUs make the chart busy, we'll add dropdown filters (e.g. select a particular brand family or segment to view in detail) or use an interactive plotly chart where you can zoom into segments. Another idea is a **line chart** per segment category: each segment's average NR/HL old vs new as points connected by a line, showing the gap between segments increased or maintained. However, the requirement explicitly says "across SKUs at a SKU level," so a granular chart is expected. We'll likely implement an interactive multi-select for segments: e.g., check boxes for Value/Core/Core+ etc., so the user can toggle which segments' SKUs to display on the chart for clarity. - **Aesthetics:** Streamlit allows basic theming; we will keep it simple and **intuitive**, using the AB InBev color palette if possible (e.g. red/gold accents) and clear section headings (e.g. "Input Parameters", "Optimization Summary", "SKU Details", "Price Architecture Chart"). We'll ensure that even though it's an MVP, the layout is clean: inputs on a sidebar or top section, outputs organized in tabs or sections, so that a user can easily navigate. - **Responsiveness:** Since this will run on a local machine (judges' laptop), we'll

ensure the UI updates are reasonably fast. The optimization might take a few seconds; we can use Streamlit's spinner or status message ("Optimizing...") to inform the user while they wait briefly.

1. **Next.js + Tailwind Front-End (Polished Version):** In parallel (or after validating the logic with Streamlit), we plan a more robust front-end using **Next.js (React)** and **Tailwind CSS** for styling. This will be a production-quality web app interface:
2. **Design & Layout:** We will create a modern, intuitive dashboard-style UI. Tailwind will let us quickly style components with AB InBev's brand guidelines (for example, using their corporate font and colors). We envision a main dashboard page where the top has input controls (sliders, inputs) and the bottom has results. Possibly use a two-column layout: left side for inputs and key outputs (like a summary card), right side for detailed table and charts.
3. **Interactivity:** React allows more dynamic interactions than Streamlit. For example, we can let the user **drag the PINC slider** and immediately see a recalculated result (if we make the solver call async or use a precomputed approximation for quick feedback). We could also allow the user to manually tweak individual SKU prices in a what-if mode: e.g. override a recommended price and see updated KPIs (this would call a re-solve with that SKU's price fixed). These are stretch features that provide a "simulation" feel.
4. **Integration with Backend:** The Next.js app will communicate with the optimization engine running on a backend. We will expose a **REST API** (or GraphQL) endpoint for optimization. Likely, we'll implement a lightweight **Flask or FastAPI server** wrapping our optimization code. The Next.js front-end will send a request with the user inputs (PINC, any custom bounds or locked SKUs) to this API. The API will run the solver and respond with the results (optimal prices and all metrics). The front-end will then render the results nicely. This separation allows scalability and integration into BrewVision (which presumably could call the same API).
5. **Dynamic Visualizations:** We will use a charting library in React (such as Chart.js or D3 or Recharts) to build interactive charts for the price architecture. The table of SKU details can be made interactive with sorting, filtering (using a component like Material-UI DataGrid or similar).
6. **Dual Language / Mobile-Friendly:** Next.js will produce a responsive web app accessible in a browser, potentially easier to integrate and more user-friendly for AB InBev teams. We can also consider multi-language support (if needed) and ensure the layout works on various screen sizes (Tailwind's utility classes help with that). This goes beyond hackathon requirements but is a consideration for a polished tool.

In short, the **Streamlit app** will serve as our development UI and proof-of-concept for demonstrating functionality quickly, and the **Next.js app** will be a more refined product interface. Given hackathon time, the Streamlit version will likely be delivered for judging (since it can be run with a single `python main.py` command easily), but we will outline and partially implement the Next.js version to showcase our plan for a production-ready tool.

User Experience Highlights: - The UI will guide the user step-by-step: select input -> run model -> view results. It will handle validation (e.g. if PINC out of range or bounds inconsistent, it will show an error or adjust). - We will include **help tooltips** or an info section explaining each input (e.g. what is PINC, what do bounds mean) and each output metric (e.g. definition of NR/HL or MACO/HL), to ensure a non-technical user can understand the interface. This addresses usability since revenue managers may not be experts in elasticity modeling. - The results will clearly indicate that all constraints are met. For example, a small textual note: " Volume –0.5% (within –1% limit), Share –0.2pp (within –0.5pp limit), MACO +5% (improved)" to explicitly show success criteria. - If any soft constraint was at risk, we might highlight "(Adjusted to maintain hierarchy)" in the table for any SKUs that had their price tweaked specifically for that. But ideally, the user shouldn't have to worry about that – the output itself will be coherent. - **Export Option:** We can add a button to **download the optimized price plan** (e.g. as a CSV file) containing SKU and new price, so the team can easily use it outside the app. We'll format it as

requested if needed (though not explicitly stated, it's a practical addition). - The UI will be designed for **fast iteration**. A user could try PINC=4%, see results, then quickly change PINC to 2% and re-run to compare scenarios (perhaps in the Streamlit version, they'd rerun and see updated table; in the Next.js, we could allow multi-scenario comparison if time permits, but likely out of scope).

By providing both an immediate Streamlit interface and planning the more complex Next.js interface, we ensure we meet the hackathon requirements for a functional web app, and also demonstrate a path to a **polished product-ready UI** that could integrate into AB InBev's BrewVision platform in the future.

Output Visualization & Reporting

Delivering clear and insightful output is crucial for adoption. Our solution will produce both **tabular reports** and **visual graphics** to communicate the pricing recommendations and their effects.

Overall Summary Table: We will prepare a top-line summary that compares the **Before vs After** at an aggregate level. This can be a small table or set of KPI cards showing: - **Total Volume (HL)** – Base vs Optimized, and the percentage change. We'll highlight that this change is within the allowed range (e.g. "-0.8% volume, OK"). - **Total Net Revenue** – Base vs Optimized, and % change. This demonstrates the revenue impact of the price changes. - **Average NR/HL** – Base vs Optimized. This effectively is a measure of price per HL (a rise in this indicates successful price increase implementation; we expect this to increase roughly by the PINC percentage). - **Total MACO** – Base vs Optimized, and % increase. This is the main objective we're improving. - **MACO/HL** – Base vs Optimized (this combines the above, showing profitability per unit volume). - **ABI Market Share** – Base vs Optimized (in % of total volume). This likely will show a slight decline (if volume drops a bit) but within the 0.5pp tolerance.

These five KPIs (Volume, NR, NR/HL, MACO, MACO/HL) are explicitly required [25]. We will present them clearly. For example:

KPI	Base (Current Year)	Optimized Scenario	Change
Volume (HL)	24,212,000	23,970,000	–1.0%
Net Revenue (COP)	5.44e12	5.60e12	+2.9%
NR/HL (COP/HL)	224,700	233,700	+4.0%
MACO (COP)	2.78e12	2.90e12	+4.3%
MACO/HL (COP/HL)	114,800	120,900	+5.3%
ABI Market Share	92.4%	92.0%	–0.4 pp

(Numbers are illustrative). The table (or set of cards) will use icons/colors (green up arrows for increases, red down for decreases) for quick visual parsing. This summary confirms that: volume drop <1%, share drop <0.5pp, MACO increased, etc., satisfying the constraints.

SKU-Level Detailed Table: This is a core output: a comprehensive list of recommended new prices and impacts per SKU [26]. The table will include columns such as: - **SKU Identifier** – name or code (e.g. "Budweiser 330ml Can"). - **Segment & Pack Type** – for context (could be separate columns or part of SKU name). - **Old Price** (PTC, in COP). - **New Price** (PTC, COP). - **Price Change** – both absolute (COP) and percentage. We might highlight if any SKU hit its bound (e.g. if new price = max allowed, we could asterisk it to indicate it was constrained). - **Volume (HL) Before** and **Volume After**, with % change. This

shows which SKUs lose or gain volume. We expect most will lose some volume if their price rose, but some might gain (for instance, if a competitor's price effectively rose more relative to an ABI SKU that didn't change much, or if we actually decreased a price for strategic reasons, that SKU would gain volume). - **NR/HL Before** and **NR/HL After** (or we can compute NR/HL on the fly as (Price/(1+markup)*(1-discount-excise))/(1+VAT) for old vs new – but effectively this will be proportional to price). - MACO/HL Before vs After – this incorporates cost as well, indicating margin per unit. If cost per HL increased 3.78% for all, then MACO/HL won't increase as fast as NR/HL, but should still increase with price. - We can also include Total MACO per SKU (Volume * MACO/HL) before vs after, though the sum of these gives total MACO. - Optionally, Market Share per SKU (i.e. SKU's share of total market) before vs after – but that might be too granular and not usually a KPI at SKU level (brand share is more common). We likely skip this at SKU level detail. - Perhaps Elasticity category** – not in table but we might annotate if a SKU was identified as highly elastic vs inelastic, to rationalize why its price moved less or more. (This could be an insight in documentation rather than in the raw table.)

The table can be made filterable (e.g. filter by brand or segment) in the UI to help users digest the info. For reporting, we will ensure it's exportable as CSV or Excel so that it can serve as the "Optimized Pricing Plan".

Visualizations: - **Price Architecture Chart:** As described, a bar chart of SKUs' price per liter before/after. We will likely present separate sub-charts for each segment to improve readability. For example, five small charts, one for each segment (Value through Super Premium), each chart showing its SKUs sorted by pack size category with old vs new price bars. This will explicitly show that, within each segment, smaller packs are priced higher per liter than larger packs (if our solution kept that), and importantly, that the Value segment's bars are all below Core's bars, etc. We can annotate the charts with segment averages. The point is to visualize that **the relative positioning of segments and packs is correct**. If any anomaly appears (like an overlap), that indicates a potential hierarchy violation which we intend to avoid. This chart essentially communicates the new **NR/HL ladder** in a more digestible way than a raw table. - **KPI Impact Chart:** Another helpful visualization is a **waterfall or bar chart** showing the contribution of different factors to MACO change. For instance, we could present a waterfall where we start from base MACO, then show the effect of price increases (positive) and volume loss (negative) resulting in the net MACO uplift. This can help explain to stakeholders *how* the profit increased (e.g. "price effect +X, volume effect –Y, net +Z"). While not explicitly required, it provides insight on whether the MACO gain came mostly from revenue per unit or from volume expansion (likely the former in a price increase scenario). - **Segment-Level Summary:** We could visualize the outcome by segment: e.g. a clustered bar chart for each segment showing base vs new Volume and base vs new NR. This would highlight if, say, the Value segment volume fell significantly while Premium maybe slightly gained volume (due to cross-effects), and how revenue shifted. It gives a high-level view of which segments the price increase was concentrated in. This is more analytical, but could be included if time permits. - **Constraint Utilization Indicators:** Possibly add small visuals or indicators showing how close to the limit each constraint is. For example, a gauge or progress bar for "Volume change: used 100% of allowed decrease" or "PINC used: 80% of 6% allowed" if user input was maximum. This communicates whether the solution is constraint-bound. In our example, if user demanded 6% PINC but we only implemented, say, 3% due to volume/share constraints, we could show "Volume drop constraint binding – could not fully use 6% PINC." However, ideally we keep within what user sets, so this might be simply confirmatory ("PINC 4% fully allocated").

Report Generation: We will ensure the outputs (tables and charts) can be easily transferred into reports or presentations: - The **data table** can be downloaded. We can also include a "Download Report" button that generates an Excel with the summary and detail, or a PDF with the table and chart. (If time is short, we will at least provide the data export; a formatted PDF report is a nice-to-have.) - The **visualizations** will be designed such that they can be screenshot or saved (Streamlit and Plotly, for

example, allow chart saving). We could programmatically save the chart image to a file when the optimization runs, which could then be included in documentation or a slide.

Documentation of Outputs: In our documentation (and possibly UI tooltips), we will define each metric: - Volume: in hectoliters, with context that 1 HL = ~100 liters. - NR (Net Revenue): revenue after discounts and excise, before VAT. - NR/HL: effectively average selling price per HL (excl. VAT). - MACO: margin after variable costs (absolute profit contribution). - MACO/HL: margin per volume, an indicator of profitability per unit (should correlate with price per HL minus cost per HL). These definitions likely mirror what the revenue managers know, but clarity helps.

By presenting the outputs in multiple forms (numeric and graphic), we cater to different stakeholders: some may trust the numbers in detail, others may prefer a quick visual check that everything aligns with strategy. Our reporting will thus ensure that **the optimized plan is transparent and justifiable**, making it easier to get buy-in for the recommended price changes. The combination of a data table with all SKU-specific changes and high-level charts meets the deliverable expectations [26] [24] and provides a comprehensive view of the solution.

Integration & Deployment

Our solution will be packaged for **easy deployment and integration**, following the hackathon guidelines for a single-command setup and considering future integration with AB InBev's BrewVision platform.

Architecture: We will separate the **optimization engine (backend)** from the **user interface (frontend)** in a modular way: - The **backend** consists of data loading, model construction, and solver execution – written in Python. We will wrap the core optimization logic in a function or class (e.g. `optimize_prices(pinc, bounds, etc)`) that returns the results. This can be invoked either by the Streamlit app or by an API endpoint. - For the **Streamlit** MVP, the Streamlit script will simply call this function and display results. - For the **Next.js** app, we will run a small Python web server (using FastAPI or Flask) that exposes endpoints (e.g. a POST to `/optimize` with JSON input containing PINC and any custom bounds) and returns JSON output (with new prices and metrics). The Next.js app (running on Node.js) will call this API. We'll ensure CORS is handled so that the front-end can fetch from the back-end if on different ports.

Single-Command Execution: We will adhere to the requirement that the judges (or any user) can start the app with minimal steps: - If using Streamlit only, we will provide a main script (e.g. `main.py`) that contains the Streamlit application. Launching `python main.py` will start the local web app. We'll include instructions (in README) to install dependencies (`pip install -r requirements.txt`) beforehand. - If delivering the Next.js front-end as well, we will ensure it's also straightforward: likely providing an `npm install` and `npm run dev` or `npm start` command. To unify this, we might include a **Procfile or a shell script** that can launch both the Python backend and the Next.js frontend concurrently. Alternatively, we can supply a **Docker Compose** setup: one service for the Python API, one for the Next.js UI. In that case, running `docker-compose up` would spin up both and they'd communicate on configured ports [27]. Given hackathon constraints, the simplest route is to default to the Streamlit app (single process), which meets the single-command criterion easily. We will document clearly how to start whichever UI is provided.

Environment & Dependencies: Our code will not require any manual environment variable configuration or secrets. All parameters (like file paths, etc.) will be configured to use relative paths or packaged data. We'll ensure compatibility across Windows/Linux/Mac by using standard libraries

(pandas, pulp, ortools, etc.) that are cross-platform. If any special system dependency is needed (none anticipated beyond maybe a solver binary for CBC which comes with PuLP), we will note it or include it. The requirements.txt will pin necessary versions to avoid conflicts.

Packaging Data: We will include the necessary data files in the repository (under a `data/` folder) or provide a download script if they are large. Since they are provided and not huge (~ a few MBs for Excel/CSV), we can include them. The app on startup can automatically load these. If performance is slow to load all data each run, we might preprocess and store a smaller structure (like elasticity matrix as a pickle), but given it's fine to load on each run (couple seconds), we likely don't need a persistent database. Simplicity is key.

Integration with BrewVision: BrewVision likely expects either an embeddable UI component or an API. We lean toward an API-based integration. Our backend can be containerized as a microservice that BrewVision calls with the necessary inputs and then it could display the output (maybe BrewVision has its own UI). Since we are also delivering a UI, AB InBev could choose to embed our UI via an iframe or integrate the logic. We will mention in documentation that **the optimization engine can run as a standalone service** – meaning AB InBev's tech team could call a function or API with elasticity and PINC input, and get optimal prices, which they can then feed into BrewVision's interface or database. We keep that in mind by designing the code to be modular (separating input parsing, optimization, output formatting).

DevOps & Deployment: For hackathon demonstration, running locally is fine. For future deployment, one could deploy the Python API on a server or cloud environment and host the Next.js app similarly (Next.js can be exported as a static app if using only client-side, or run on Node for SSR). We'll ensure that migrating to a production environment only requires updating data sources (e.g. connecting to a production database of prices instead of the Excel) but not code logic changes.

Error Handling & Logging: We will include basic error handling. For instance, if the solver fails to find a solution or if input values are out of bounds, the backend will catch exceptions and return a meaningful error message (which the UI can display to the user). We'll also log key events (like "Optimization started with PINC X" and "Solution found with MACO Y") to console or a log file for debugging. In a simple Streamlit scenario, print statements in the backend can serve to debug in console.

Testing Deployment: We will test the deployment process on a fresh machine to ensure the instructions are correct – e.g., clone repo, install requirements, run main.py, and it works without additional tweaks. This check is crucial to avoid any environment-specific hiccups during judging. If using Docker, we will build the image and run it to ensure it encapsulates everything (this might not be required if not explicitly asked, but can be a backup option for ease).

By following these integration and deployment practices, we ensure our solution is **plug-and-play** for the judges and can be easily transitioned to AB InBev's environment. The single-entry script and optional Node start script adhere to the guidelines [28]. Moreover, the separation of concerns in our design (frontend vs backend) means that if AB InBev wants to use only the engine and build their own UI in BrewVision, they can do so by calling our code. Conversely, if they like our UI, they can integrate it with minimal changes.

Overall, the deployment will be streamlined, requiring at most a few commands to get up and running, and the architecture will support future scaling or integration.

Demo Script

To illustrate how a user (e.g. a Revenue Manager) would use our pricing optimizer, below is a step-by-step walkthrough of a typical session. This serves as a **demo script** and sanity check of the end-to-end functionality:

1. **Launch the Application:** The user opens a terminal and runs the app with a single command (for example, `python main.py`). The application starts a local web server and the user navigates to the provided local URL (Streamlit usually opens a browser automatically). The interface titled **"ABI RGM Pricing Optimizer"** loads, displaying input controls on the left and an empty results area on the right.

2. **Set PINC and Constraints:** The user sees a slider labeled "**Total Portfolio Price Increase (PINC)** (%)" with a default value (say 3.0%). They adjust it to their desired scenario – for instance, they choose **5.0%** to simulate a substantial price increase scenario (close to the upper 6% limit). Just below, there are fields for **"Global Price Change Bounds"** with defaults "Min: –300" and "Max: +500" (in COP). The user decides to tighten the lower bound to 0 (no price cuts) because they only want to consider price increases in this run. They leave the upper bound at +500. (If using the Next.js UI, this might be done via a form with the same fields; in Streamlit, it could be number_input widgets.)

3. **Advanced Input (optional):** The user could expand an "Advanced options" section. For example, there might be a multi-select list of brands where they can enforce "no change" on certain strategic SKUs. In this demo, suppose the user selects one item: they choose to **freeze the price of "Aguila 330ml NRB"** (perhaps a flagship core SKU) by setting its min and max change to 0. (Our UI would allow this by listing SKUs and a checkbox or input for custom bounds; if not, assume default all are free within global bounds.)

4. **Run Optimization:** The user clicks the **"Optimize Prices"** button. Immediately, the interface provides feedback: a status message "Running optimization – calculating optimal prices..." appears. In the background, our engine receives the input: PINC=5%, bounds [0, +500], freeze Aguila 330 NRB. The solver constructs the model and begins solving. Given the problem size, within a few seconds we have a result. The status message disappears and the results populate on the UI.

5. **View Summary Results:** At the top of the results section, the user sees a **Summary** panel. For example:

6. **Volume:** 99.0% of base (–1.0%). A green check icon is next to it, indicating this is within the –1% limit (exactly at the limit in this case, since 5% PINC likely pushed volume drop to the allowed maximum).
7. **Net Revenue:** +3.8%. This is the increase in net sales revenue.
8. **NR/HL:** +5.0%. This matches roughly the 5% PINC input, confirming the average price per liter rose 5%.
9. **MACO:** +4.5%. An arrow shows upward, indicating profit increased by 4.5%.
10. **MACO/HL:** +5.6%. Margin per unit went up a bit more than price per unit, possibly due to mix effects.
11. **ABI Market Share:** –0.4 pp. It shows share dipping from, say, 92.4% to 92.0%, which is a 0.4 percentage point drop, within the 0.5 limit (also flagged with a green check).

These high-level numbers tell the user that the plan meets targets: volume dropped by exactly 1% (binding that constraint), share loss 0.4 (close to limit but fine), and MACO is improved. The user is satisfied that constraints are respected (perhaps noticing volume hit the lower bound, which is expected for a high PINC scenario).

1. **Examine SKU Price Recommendations:** The user scrolls to the **SKU Details** table. They see each SKU listed row by row. They notice:
2. Many of the Premium and Super Premium SKUs have received close to the full +6% price increase (since the overall PINC was 5%, some high-end SKUs might be raised slightly above 5% to compensate for lower increases elsewhere). For example, "Club Colombia 330ml" (Premium) old price 2000, new price 2120 (+6%). Volume for it fell by ~4%, but since it's premium (inelastic), that's acceptable.
3. Core segment SKUs like "Aguila 330ml NRB" (which they froze) show new price = old price (no change, as enforced). Volume for Aguila 330ml actually increased slightly (+0.5%), likely because other beers in Core raised price, making Aguila relatively cheaper, so it gained some volume via cross-elasticity. The table might highlight that with a green volume change.
4. Another core SKU, "Aguila 500ml CAN" (Large Core) maybe was allowed to increase a bit, say +2%. Volume dropped slightly. Possibly the optimizer left value segment almost untouched to protect volume: e.g. a "Value 750ml bottle" might show only +1% price, because value consumers are very price sensitive (elastic).
5. Value segment SKUs mostly show minimal increases (1–2%), and one might even show 0% if volume drop was a concern.
6. The table confirms **hierarchies**: the user spots that every Premium SKU's NR/HL is higher than every Core SKU's NR/HL. If they sort the table by NR/HL (new), all Super Premium items are at top, then Premium, down to Value at bottom, with no intermixing – proving the ladder constraint held. Similarly, they could filter to just Small packs and see those have higher per-liter prices than Regular of the same segment.
7. They also notice some interesting cross effects: e.g. a competitor brand "Poker (Competitor) 330ml" in the table (if we display competitor SKUs too, which we might for completeness) shows that its volume went up by, say, 3%. It's not an ABI SKU, but including it demonstrates competitor gains. Its price column would show no change (since we didn't alter competitor price), but volume column shows +X%. This explains where ABI lost share.
8. Key SKUs: The user checks a high-volume Core SKU (besides Aguila) – say "Poker 330ml CAN" (if that's ABI) – and sees it got perhaps a +3% price. Volume might drop ~1%, but because of its size, that contributed a chunk to the volume decline.
9. The table might be color-formatted: e.g., price increases in green, volume drops in red. This helps the user quickly see patterns: lots of green in price column (as expected with PINC positive), mostly red in volume change (volume down for most SKUs, except the frozen or low-increase ones which might have green volume because they gained share).
10. The user can search within the table for a particular brand or segment to inspect specifics.
11. **Visualizing Price Architecture:** The user then looks at the **Price Architecture** chart. It might be a segmented bar chart by segment:
12. For "Value" segment: the chart shows old vs new price per liter for each Value SKU. They see all value SKUs had small increments, and their absolute price per liter remains lowest.
13. For "Core" segment: they see a noticeable gap above Value. For instance, Value max might be 5,000 COP/HL, Core min is 5,500 COP/HL after optimization – so gap maintained.
14. The chart for "Premium" vs "Core+" vs "Super Premium" might show those all moved roughly +5-6%, so their relative spacing remained (maybe even widened slightly).

15. They also note **pack size effect**: within the Core segment chart, the **Small** (say 250ml bottle) bar is highest, Regular (330ml) a bit lower, and Large (750ml) lowest per liter, both before and after. The new bars maintain the ordering (small pack still highest per liter). The differences might have slightly changed magnitude but not ranking.
16. This gives a clear visual confirmation that the solution **didn't violate any brand/pack positioning**. If needed, the user could toggle some chart options to drill down (in Streamlit, less interactive than in Next.js; but we could have one combined chart with color grouping by segment).

17. The user is pleased to see that, for example, "Super Premium" beers are still far above "Core" beers in price/HL – in fact, maybe even more so after the increase, which aligns with a strategy to premiumize.

18. **Interpreting the Outcome:** The user concludes:

19. The optimizer concentrated price increases on Premium/Super Premium SKUs (maximizing revenue where volume impact is low) and kept Value segment prices almost flat (protecting volume and share). This matches intuition and is now quantified.
20. The overall plan yields a +4.5% MACO improvement with only a 1% volume loss and negligible share loss – a good trade-off.
21. All hard constraints are exactly met or slightly under: volume fell by just 1.0% (at the limit), share fell 0.4pp (<0.5), and PINC 5% was fully utilized. This suggests the solution is **constraint-bound** by volume drop in this scenario, which is expected for a high PINC.
22. The user might experiment further: say, reduce PINC to 3% and run again to see if volume drop is much less and which constraints bind (likely none bind strongly at 3%, so solution might spread increases more evenly). The app allows them to do that quickly if desired.
23. They could also try a scenario with allowing some price cuts (setting a negative min bound) to see if any SKU would actually get a price drop (perhaps not if PINC positive, but if PINC was small, maybe the optimizer might drop a price of a very elastic SKU to gain share and make up profit elsewhere – an interesting scenario).

24. For demonstration purposes, our script could show just the one scenario, but it's easy to vary inputs.

25. **Exporting & Next Steps:** Satisfied, the user clicks "Download Results". The app generates a CSV file "Optimized_Prices.csv" where each row is SKU, old price, new price, etc. They save this for reference. The user notes they can integrate this into their BrewVision tool or share with finance for review. If this were a real deployment, they might also use the insights (like which SKUs hit bounds) to consider adjusting bounds or strategy (e.g., maybe they see a certain SKU would have wanted more than +500 COP – indicating there's room for an even larger increase if allowed).

26. **Closing the App:** The user ends the session. The entire process (input to results) took maybe 5–10 seconds of computation and a few minutes of exploration. The interface was responsive and informative, allowing them to confidently derive a pricing strategy scenario.

Throughout the demo, we ensured to highlight how the tool meets the expectations: - It took the user's input (PINC, bounds). - It produced optimal SKU prices that improve MACO and listed all impacts. - It respected the constraints (the user could see none were violated). - It provided a dynamic visualization of the SKU price architecture after the change. - It allowed for quick scenario adjustments (which a revenue manager would appreciate for planning "what-if" situations).

This script can be used live to demonstrate the app or as a guiding narrative for our documentation. It shows the **practical workflow** and how the analytics translate into actionable outputs. It also helps validate that our design (in terms of UI elements and model outputs) is user-friendly and aligned with the problem needs.

Repository Structure & Code Organization

We will organize the project repository in a clear, logical manner to separate different components and make the code easy to navigate. Below is the planned structure:

```
├── README.md
├── requirements.txt
├── data/
│   ├── Elasticity_Matrix.csv
│   ├── Price_List.xlsx
│   ├── Sellout_Train.xlsx
│   ├── Sellout_Test.xlsx
│   ├── Sellin.xlsx
│   └── IHS.xlsx
├── src/
│   ├── data_preparation.py
│   ├── elasticity.py
│   ├── optimizer.py
│   ├── constraints.py
│   ├── solver.py
│   └── output_formatter.py
├── app/
│   ├── main.py              # Streamlit app (entry point)
│   ├── backend_api.py       # (Optional) Flask/FastAPI app for Next.js
integration
│   └── frontend/            # Next.js frontend project (if included)
├── tests/
│   ├── test_data.py
│   ├── test_optimizer.py
│   ├── test_constraints.py
│   └── test_end_to_end.py
└── docs/
    └── *.md (documentation pages for MkDocs)
```

Top-level files: - `README.md` – Contains an overview, setup instructions, and basic usage examples. This is the first thing a user/judge will see, so it will concisely explain how to run the app (e.g. "`python main.py` launches the Streamlit web app on localhost, default port…" or if using Node, the appropriate command). It will also list the project structure for reference. - `requirements.txt` – Lists all Python dependencies (e.g. `pandas`, `pulp` or `ortools`, `streamlit`, `flask` (if needed), etc.). We'll pin specific versions known to work to avoid environment issues.

Data Directory (`data/`): All provided datasets are stored here. Our code will load from this folder. We included them in the repo for completeness; if that's too heavy, we would mention any needed

download. But since they are provided, we assume it's fine. - We will treat these as read-only inputs. The optimization results could be saved to a separate outputs directory if needed, but mostly we output through the UI.

Source Code (`src/`): This contains the core logic, separated by functionality: - `data_preparation.py` – Functions to read the raw files and create cleaned DataFrames or dictionaries. E.g. a function to load price list and return a dict of base_price, markup, etc. Also a function to load base volumes by SKU from Sellout data (aggregating 2024), and to compute base MACO from Sellin data. Essentially ETL tasks to get our input parameters ready. - `elasticity.py` – Functions related to the elasticity matrix, e.g. loading it and perhaps converting it into a structure like `elasticity[target_sku][other_sku] = value`. Possibly also a function to retrieve all elasticities for a target (to build volume constraints). - `optimizer.py` – This will contain the `optimize_prices()` function which orchestrates building the optimization model and solving it. Inside, it will likely call sub-functions or use classes from `constraints.py` and `solver.py`. For example, it might: - prepare data (via data_preparation), - initialize the optimization model (using PuLP or OR-Tools), - call `constraints.add_hard_constraints(model, variables, data)` and `constraints.add_soft_constraints(model, variables, data)`, - call `solver.solve(model)` to run the solver, - then format the results for output. - `constraints.py` – Defines functions to add each set of constraints to the model. For example, a function `add_volume_constraints(model, vol_vars, price_vars, elasticity)` that adds $Volume_i(new) = Volume_i(base) + \dots$ for all i. Similarly, functions for add_share_constraint, add_volume_target_constraint, etc. This modularity helps because we can test each constraint function individually (in `test_constraints.py`) and ensure they correctly represent the math. - `solver.py` – Contains the interface to the solver. For PuLP, this might just call `model.solve()`. For OR-Tools CP-SAT, we'd set up the CP-SAT solver, add variables and constraints to it (maybe not needed if we built in PuLP then exported LP file, but likely we'll directly use one method). It can also handle toggling between a linear vs quadratic solve, or any solver-specific settings (time limit, etc.). It will also contain logic for handling infeasibility (like check solver status, if infeasible, perhaps try relaxing something or return a message). - `output_formatter.py` – Functions to take the solver's output and compute all metrics and tables needed for display. For example, given new prices and volumes, compute NR, MACO per SKU, segment averages, etc., and produce the pandas DataFrame used in the Streamlit app or the JSON for the API. Essentially turning raw results into presentation-ready data. This separation allows easier formatting changes or reuse between Streamlit and Next.js outputs.

We might not strictly need every file separate (some could be merged), but this structure keeps things organized by purpose, which is helpful for maintainability and clarity.

Application (`app/`): - `main.py` – This is the entry point for the Streamlit UI. It will import from `src/` modules. The structure of main.py will be something like: load base data (could be at startup or cached), display input widgets, on button click call `optimize_prices(pinc, bounds, user_overrides)` from `optimizer.py`, get results, then use Streamlit to display the summary, table, and charts. We'll keep Streamlit-specific code here so that the src/ logic is decoupled (which also allows unit testing of src without needing Streamlit). - `backend_api.py` – (If implementing Next.js integration) This might be a small FastAPI application. We'd include endpoints like `POST /optimize` that parses JSON input, calls `optimizer.optimize_prices` internally, and returns JSON output. We would guard this with if `__name__=='__main__'` to run it as a script (maybe via `uvicorn`). This file might only be used if we are actually demonstrating the Next.js front-end; otherwise it's optional. - `frontend/` – This folder would contain the Next.js project (with pages, components, etc.) if included. For hackathon submission, including a whole Node project might complicate things, so we may include it only if fully working and easy to launch. If included, `README` will have instructions (like `cd app/`

`frontend && npm install && npm run build && npm start`). The Next.js project itself will have its own structure (pages, etc.), which we won't detail here, but it will be a self-contained client.

Tests (`tests/`): We will write unit and integration tests to ensure everything works correctly: - `test_data.py` – Tests for data loading and assumptions. For example, verify that base volumes sum up correctly, that the sum of VIC+VLC from Sellin equals base VILC used, etc. Also test that elasticity matrix values are within expected ranges (0 to +something for cross, negative for own) to ensure no weird input. - `test_optimizer.py` – Test the optimize_prices function on a small mock scenario. We might create a toy elasticity matrix with 2 SKUs and known outcomes to see if the solver picks the right solution (for example, one SKU elastic, one inelastic, check that the inelastic one gets the bigger price increase). - `test_constraints.py` – For each constraint-adding function, we can set up a small model and ensure the constraint behaves. E.g., feed a known situation to `add_share_constraint` and see that the resulting constraint equation matches expectation. - `test_end_to_end.py` – A high-level test running the whole pipeline with actual data for a trivial case (maybe PINC=0 scenario should output identical prices and base volumes, which is a good sanity test: when PINC=0, the optimal solution is to do nothing, so we expect ΔP all zero, and output metrics equal base metrics). Another end-to-end test might use PINC=some small value and check that output constraints are within limits. We'll also test the Streamlit interface logic by simulating inputs (if possible) or at least test that no exceptions in optimize_prices for given real data.

We will integrate these tests into the repository such that running `pytest` (if we include pytest) will execute them. This not only helps our development but also showcases to judges that we have a robust approach (per AI scoring guidelines emphasizing testing and code quality).

Documentation (`docs/`): We plan to use **MkDocs** to create a documentation site. In this folder, we will have markdown files covering: - Introduction.md – overview of the problem, our approach (similar to sections of this plan, but refined for documentation form). - Data.md – description of the data sources and any processing. - Methodology.md – detailed explanation of the model (basically a cleaned-up version of Optimization Model + Constraints + some Solver notes, including equations). - UserGuide.md – how to use the app (could incorporate the demo script as a use case example). - possibly Architecture.md – how the code is structured and how to deploy, for developers. - We will configure `mkdocs.yml` to generate a site from these. The README might suffice for quickstart, but MkDocs will host the in-depth content including any visuals or diagrams if we add. - If possible within time, we might deploy this documentation via GitHub Pages for easy access, but at least it will be in the repo.

Version Control & Branches: We will do development in a dev branch and merge to main. The submission will use the `main` branch as required [29], cleaned of any debug code. We will ensure commit history is organized and messages are clear, reflecting our methodology.

This structure ensures **separation of concerns**: - Data/logic vs UI vs tests vs docs are all separated. - It will be easy for someone to find, for example, where constraints are defined or where the Streamlit app lives. - Adding new features (like another constraint or a different UI) would be straightforward due to this modular design.

Finally, we will double-check that no large unnecessary files are in the repo (to keep it lightweight), and that all secrets (not applicable here) or personal info are removed. The repository will be structured to impress both the AI code quality checker and human judges: clear, documented, and with evidence of good software practices.

Documentation Plan

We recognize that a well-documented project is critical for understanding and maintaining the solution. We will prepare comprehensive documentation targeting both technical judges and future users. The documentation will be structured in multiple layers:

1. In-Code Documentation: We will include clear comments and docstrings in the codebase: - Every function in the `src/` modules will have a docstring explaining its purpose, inputs, and outputs. For example, `optimize_prices()` will have a docstring describing the algorithm and assumptions. - Complex sections (like building the volume constraints or computing share) will have inline comments referencing the formula or constraint they implement, possibly with citation to the problem statement (to show we followed it exactly). For instance: `# Constraint: Total Industry Volume drop <=1% (Ref: Round2 Instr. pg8)`. - We will also use consistent naming to make the code self-documenting (e.g. `pinc_target`, `volume_base`, `elasticity_matrix`, etc. are intuitive). - This internal documentation aids any developer or judge reading the code to quickly grasp what each part is doing and verify correctness against the specs.

2. README.md: The README will serve as a quickstart guide: - It will start with a **project overview** (a short description of Round 1 and Round 2 goals, so a newcomer understands context). - Then **setup instructions**: how to install dependencies, how to run the app (with examples for both Streamlit and Next.js paths). - It will mention at a high level the tech stack used (Python, solver library, Streamlit, etc.) and how the repo is organized (a summary of the structure we described). - This is the first point of contact for judges, so it will be concise but informative, ensuring they can get the app running and know what to expect. - We will also include any notes about platform specifics (e.g. "Tested on Windows 10 and Ubuntu 20.04, Python 3.9") and troubleshooting tips (though we aim to require none).

3. MkDocs Site: We will use **MkDocs** to create a multi-page documentation site that provides a deep dive: - **Introduction:** A page describing the business problem and our solution approach in lay terms. It will reiterate success criteria and how our solution addresses them. - **Methodology:** This page will detail the Round 1 elasticity approach (briefly) and mainly the Round 2 optimization methodology. We will include the key formulae (MACO, NR/unit, etc.) and explicitly list the constraints (perhaps in a list just like in this plan, with maybe a checkmark indicating we adhered to each). This section benefits from the mathematical clarity – we may include a few equations or inequalities for clarity (in LaTeX if MkDocs allows, or images if not). We will also describe how we linearized and solved the model. Essentially, a written form of the "Optimization Model" and "Constraints" sections, but even more polished and potentially simplified for readability. - **Data & Assumptions:** A page listing the data sources used and key assumptions (like cost inflation, competitor static). We'll provide a short description of each dataset and how it was used, akin to a data dictionary but focusing on relevant fields. Also mention any data cleaning or transformations we did. - **User Guide (Using the App):** This page will contain step-by-step instructions (similar to our demo script) on how to use the web application. We will include screenshots of the UI – for example, showing the input form, the output table, and the chart. We'll annotate these images to explain what each element means. We'll also provide guidance: e.g. "If you set a very high PINC, watch the volume constraint metric – it may hit the limit." This helps users interpret results. Essentially, this is a training document for end users. - **Results & Analysis:** We may include a section discussing the output of our model on the given data – not as required by the hackathon, but to demonstrate insight. For example, we might note "The optimizer tends to focus increases on premium brands – this aligns with intuition that premium consumers are less price sensitive. Value brands see minimal increases to protect share." We can include a chart or two here as well (like segment-wise price changes). This shows we not only built a tool but also can derive business insights from it. - **Development & Future Work:** A page targeted at developers (or judges interested in our process).

Here we cover the repository structure (some content from README), how to extend or modify the model, and discuss any additional enhancements we considered (essentially summarizing the Future Work section). We will also mention how we used GenAI if applicable (the hackathon asked how GenAI was leveraged – we can mention we used tools like GitHub Copilot or ChatGPT for brainstorming or documentation assistance, if we did, to satisfy that question). We'll address how one could incorporate further complexities (multi-country, dynamic competitor pricing, etc.), showing that we thought beyond the current scope. - **Testing & Quality Assurance:** Briefly document our testing strategy and perhaps how to run tests. Also mention any limitations known (e.g. the linear approximation assumption).

We will configure **navigation** in MkDocs such that the sections flow logically. We will make sure to incorporate the **citations and references** to hackathon documents in the methodology section (just as we have done here), to show that our design is grounded in the given requirements. The MkDocs will also allow including the required formulas nicely formatted.

4. Docstring Generation & API docs: If appropriate, we could generate an **API documentation** page using something like Sphinx or MkDocs plugins that pulls docstrings from our code. Given the scope, this might be overkill, but at least we will ensure the code is written such that if someone reads it or generates docs from it, it's understandable. If time permits, we might add an "API Reference" section in the docs that lists key functions/classes and their purpose, but it's lower priority compared to the conceptual documentation.

5. Diagrams & Visual Aids: We will include some diagrams in documentation for clarity: - Possibly a **flowchart** of the optimization process: showing how data flows in, solver runs, and outputs flow out to UI. - Maybe an **architecture diagram** illustrating the Streamlit vs Next.js dual approach (front-end-backend separation). - A chart showing e.g. the base vs optimized price ladder (as in results analysis).

These help non-technical stakeholders grasp what the tool is doing internally.

Maintaining Documentation: We plan to keep docs updated as we code. We'll cross-check that every constraint or formula in code is described in docs, and every assumption in docs is actually implemented in code. This prevents divergence.

Delivery: We will likely put the docs on a `gh-pages` branch using MkDocs to generate a site (if allowed), or at least supply the markdown files in the repo (which can be browsed on GitHub). Judges can view the markdown or run MkDocs themselves to view the site. We'll mention in README how to build docs (like "run `mkdocs serve` to view documentation locally").

Slide Presentation: While the hackathon said no need for separate PPT, our MkDocs content (especially Introduction, Methodology, Results) can double as material to talk through during any presentation or Q&A. We'll ensure it's written clearly enough that it could be read standalone or narrated.

By delivering thorough documentation, we ensure not only transparency for judging but also that AB InBev's team could pick this project up and understand both the **"why"** and **"how"** behind it. It demonstrates professionalism and helps maximize the AI documentation score [30] as well. Our documentation plan aligns with those criteria, covering methodology, assumptions, code structure, and usage.

Testing Strategy

Quality assurance is a major part of our build plan. We will implement a multi-level **testing strategy** to verify that the model and app work correctly, adhere to constraints, and produce reliable results:

1. Unit Testing (Function-Level): - We will write unit tests for all key functions in the `src` modules. For example: - **Elasticity Loading:** Test that the elasticity matrix is read correctly and that own elasticities are negative and cross elasticities are positive within reasonable ranges (0 to ~+1 or +2 as per expectations [16]). We can pick a few known SKUs from the data and assert their own elasticity is negative and at least a certain magnitude, and that they have the required number of cross-elasticities (≥2 ABI and ≥2 competitor as per Round1 criteria [31]). - **Volume Calculation:** Given a small mock elasticity matrix (say 2 SKUs), test that our linear volume formula computes expected values. For instance, if SKU A has own E = –1 and cross E to B = +0.5, we can manually compute what volume changes we expect for certain price changes and check the function's output. This ensures our implementation of the volume constraint formula is correct. - **Constraint Assembly:** For each constraint builder function, we can test that it adds the intended constraint. We might create a dummy optimization model object in tests (depending on solver, maybe use PuLP's LpProblem as a container) and then inspect its constraints. For example, after calling `add_volume_target_constraint`, we can retrieve the constraint expression and verify it matches `Vol_new_total <= 1.05 * Vol_base_total` and `>= 0.99 * Vol_base_total`. - **Objective Calculation:** Test the objective function calculation for a simple scenario. For example, if we input a scenario with one SKU, known base volume and price, test that the profit calculation (NR – cost) matches manual calculation. Similarly, test MACO_base computation from Sellin data to ensure we correctly sum NR – VILC. - **Price Ladder Check:** We might test the logic that ensures hierarchy. For instance, feed in a set of SKUs with different segments and assigned prices, then run a function that checks ladder ordering. This is more of a validation than production code, but we can include a function `check_hierarchy(prices)` that returns True if ladder holds. Use it in tests and maybe even in the app after optimization as an assertion.

- We will use a framework like **pytest** for these unit tests. We aim for coverage on all critical logic (model constraints, calculations).

2. Integration Testing (Multi-Function/Module): - **Solver Integration Test:** Run the optimization on a **very small synthetic dataset** where we can predict the outcome. For example, craft a scenario with 2 ABI SKUs and 1 competitor SKU: - SKU1: higher margin, low elasticity; SKU2: lower margin, high elasticity. - If PINC is moderate, we expect SKU1 gets most of the increase. We can solve this tiny model by brute force and then see if our solver's output matches. - We can disable some constraints if not needed to test specific focus (like temporarily ignore share constraint if irrelevant in that scenario). - **End-to-End with Real Data (Dry Run):** Without the UI, call `optimize_prices()` with a typical input (e.g. PINC 2%, default bounds) on the full dataset. Check that: - It returns a result (doesn't crash, solver finds a solution). - All hard constraints are satisfied in the result. We'll programmatically verify: - Compute total volume new vs base, assert within ±1% [18]. - Compute share new vs base, assert drop ≤0.005 [20]. - Check pricing steps: all new prices minus old prices is multiple of 50 (modulo 50 = 0). - Check no price beyond bounds: new between base-300 and base+500 for all. - Check MACO_new > MACO_base. If any assert fails, we have an issue to fix. - We'll also verify soft constraints in this test: for each segment pair, ensure NR/HL ordering holds (we can allow equality, though ideally strict inequality as per data precision). - This test acts as a final validation that our optimization meets the spec on actual input. - **UI Integration Test:** We will not automate a browser test (due to time), but we will simulate key parts: - Test that the Streamlit `main.py` can run without error. Streamlit's testing is limited, but we can factor out logic. For instance, have a function that given an `OptimizationResult` object will produce the dataframe and charts. Test that function with a sample result to ensure it handles various data (like

formatting percentages properly, no division by zero, etc.). - If possible, use Streamlit's testing hooks or simply run `streamlit run main.py` in a CI environment to ensure it launches (though not easy to automate UI interactions). - For the API (if built), use **requests** in a test to POST a sample request to our Flask app (running in test mode) and assert the JSON response has the expected structure and values within constraints. This ensures the backend API is working. - We'll also have a test that executes the end-to-end pipeline with PINC=0 as mentioned. That scenario should result in **no price changes**. So we expect ΔP all zeros and outputs equal base. This is a good sanity check (like a unit test for identity scenario). We'll assert volumes didn't change, share didn't change, MACO_new equals MACO_base, etc. If our solver is correct, with PINC=0 the best solution is indeed do nothing (MACO stays same but that meets ">= base MACO"). The solver might trivially achieve that. This test confirms that the solver doesn't introduce any weird change when not needed.

3. Performance Testing: - We will test how long the solver takes on the full dataset. We can instrument the code to print or log the solve time. In a test scenario or using a small script, we'll try worst-case inputs (PINC=6% which likely is hardest because constraints active). We want to ensure it solves in acceptable time (preferably under e.g. 30 seconds). - If we find performance issues, we might optimize by reducing constraint count or using a better solver. Since we plan OR-Tools CP-SAT, which is quite fast, we expect <5 seconds solve time. We will measure it. - We will also test memory usage informally (ensuring we're not holding extremely large structures; but 153x153 elasticity matrix is fine in memory).

4. User Acceptance Testing: - Although not formal unit tests, we will manually test various **use cases** through the UI: - Typical scenario (PINC mid-range, see outputs). - Edge scenarios: PINC = 0 (no increase), PINC = 6 (max). Check outputs carefully for constraint adherence and that the UI displays everything without error (like no chart failures or division by zero in percent calc if base volume = 0 for some tiny SKU – we should handle such edge as well, maybe exclude SKUs with no base volume from percentage calc). - Try adjusting bounds, e.g. set a very tight bound on one SKU and see if the solver honors it (its price should stay at the bound if that's optimal or required). - If possible, simulate an extreme case: remove volume constraint and run 6% PINC to see what would happen (just for curiosity, to ensure our model can technically handle beyond constraint if allowed – not for final, but to test the binding of constraints). - We will also have someone else on the team use the UI (fresh eyes) to ensure it's intuitive and that all labels make sense.

5. Testing Constraint Satisfaction Programmatically: We might include a post-optimization function `verify_solution(solution)` that checks every constraint (hard and soft) and returns a pass/fail or list of any violations. This can be used in tests (and even could be displayed in UI in debug mode). This double-check uses actual results to ensure nothing was overlooked in modeling. For example, it recalculates share and compares to base share with threshold, etc. We will run this in our test suite for a solved solution.

6. Continuous Integration: If time permits, we'll set up a simple CI (e.g. GitHub Actions) to run tests on each commit. This ensures we don't break anything as we refine code. Even without full CI, we will frequently run our test suite during development.

7. Linting and Code Quality: We will use a linter (like flake8 or pylint) to enforce consistent style and catch simple errors (unused variables, etc.). While not exactly testing, this improves code reliability. The hackathon AI scoring looks at structure and documentation [30], so having a clean codebase with no obvious lint issues helps.

Through these layers of testing, we aim for a robust final product: - The algorithm will be **validated** against known scenarios (ensuring correctness). - All the hackathon constraints will be

programmatically **verified** to be satisfied in outputs. - The UI and user experience will be **smooth and bug-free**, as we'll catch issues like mis-labeling or crashes during manual tests. - Our test suite will also serve as a form of documentation: new developers can run it to see that all components function as intended.

By the time of submission, we should be confident that our solution not only works for the provided data but is also resilient to minor changes or expansions, thanks to these tests. This disciplined approach to testing will be highlighted in our methodology and is an important part of delivering a quality solution, in line with the hackathon's emphasis on code quality and reliability. Each passed test gives us and the judges confidence that the **optimizer is doing exactly what it's supposed to – no more, no less**.

Potential Risks & Mitigation

Despite careful planning, every project has uncertainties. Here we outline potential risks and challenges, along with our strategies to mitigate them:

Risk 1: Infeasible Solution for High PINC Inputs. If a user requests a very high PINC (e.g. 6%) and the constraints (especially volume drop $\leq$1%) make it impossible to allocate that fully, the solver might declare infeasibility. This could occur because our model currently would try to enforce exactly 6% average increase while volume cannot drop more than 1%, which may conflict as discussed. **Mitigation:** We have a few approaches: - We will implement a **graceful degradation**: if the solver finds no feasible solution at the exact PINC target, we can relax the PINC constraint to become an inequality ($\leq$ target) and re-solve. This way, the solver will use as much of the PINC as possible without breaking other constraints. We will then inform the user: e.g. "Only 3.5% out of 6% PINC could be implemented without violating volume/share limits." This communicates the outcome rather than failing. Technically, we can do this by initially setting PINC as equality, and if no solution, change it to $\leq$ and add an objective penalty to minimize unused PINC. - Alternatively, we could allow the volume constraint to ease if absolutely needed (since 1% drop vs maybe 2% drop might be acceptable if user insisted on 6% PINC). But since it's a "hard" constraint, we prefer not to break it. So the first approach is better: do not force full 6% if not feasible. - We will test the extremes to confirm whether infeasibility arises. If it does, the above logic will be included. If not (maybe the solver finds a way to honor everything by heavy cross-subsidization among SKUs), then fine. But likely, as we suspect, at 6% the model will saturate volume drop constraint and might not reach exactly 6%. So our UI will be prepared to handle and explain that scenario.

Risk 2: Elasticity Model Accuracy. The optimization is only as good as the elasticity inputs. If some elasticity values are off or not truly linear, the recommendation might be suboptimal or even counter-intuitive in reality. For example, if cross elasticity between two SKUs is overstated, the model might under-price one SKU to gain volume that wouldn't actually materialize. **Mitigation:** - We will incorporate a bit of **conservatism**. For instance, we can impose bounds on elasticities usage: Round 1 criteria said own elasticities should be between 0 and –5 [16], cross between 0 and +5 (though most of ours likely smaller). We can examine the elasticity matrix and possibly **cap extreme values** to avoid crazy outcomes. If we found any cross elasticity above, say, +2 that seems unrealistic, we might clamp it when building the model. This prevents the optimizer from exploiting an probably spurious relationship. We have to be careful modifying data, but slight rationalization is okay. - We will also run **sensitivity tests**: after we get an optimal solution, we can simulate a ±10% change in a few key elasticity values to see if the solution changes dramatically. If yes, it means that price recommendation is fragile. We could then adjust by, say, not going all the way to a bound for that SKU's price. However, given hackathon context, we might not iterate that deeply, but we will mention these considerations in documentation for

realism. - Business logic overlay (soft constraints) also helps mitigate some elasticity issues: e.g. even if elasticity suggested a Value beer could take a big increase (maybe data noise), the segment ladder constraint would stop us from raising it above Core. - If time allows, we could integrate a **scenario analysis** mode where user can tweak elasticity for a critical SKU and see impact. But that's an enhancement beyond initial scope.

Risk 3: Overfitting to Data / Lack of Generality. The solution might be tailored too specifically to current data (e.g. using parameters like 0.56 which might not hold if conditions change). If AB InBev later uses the tool in a slightly different context (another country or updated data), it should still work. **Mitigation:** - We keep the model flexible: most constraints are relative (percentages) so they scale. The code is not hardcoded to specific SKUs; it reads from data, so new SKUs or changed volumes will flow in naturally. - The industry volume factor 0.56 might differ by market. We will clearly parameterize it (maybe as a constant in a config or easily changeable variable). Same for volume ±1%, share 0.5%. If AB InBev decides to allow –2% volume, they can change one line in constraints config. - We will mention in documentation how to adjust these parameters, making the tool more general. - We can also test on a subset of data (like simulate if one competitor had different share) to ensure model still behaves.

Risk 4: Solver Performance Bottleneck. There is a possibility that the MILP could take longer than expected to solve, especially if the search space is large (some SKUs with wide bounds and trade-offs might cause lots of branching). **Mitigation:** - Use solver time limits: we can set, say, a 30 second time limit. If it hasn't found proven optimum, it will return the best found solution. Because our problem is not extremely large, CP-SAT might already find optimum quick but struggle to prove it – we can accept a near-optimal solution if within a tiny gap (we can check solver's gap). Typically CP-SAT often finds optimum within seconds for these sizes. - If needed, we can simplify the model: e.g. if certain SKUs are negligible (very low volume or revenue), we could fix their price change at, say, average PINC to cut variables. This reduces complexity. We will see if needed from test runs; if solve time is already <5s, no need. - We will also ensure to run in **release mode** – e.g. OR-Tools CP-SAT uses multiple threads; we can allow multi-threading (with a note that it might use up to all CPU cores, which is fine for a short burst). - Given the hack environment, having a snappy demo is crucial, so we'll tune down complexity if any delay is noticed.

Risk 5: Minor Numerical Issues. Linear programming can have numerical precision issues if coefficients are very large/small. For example, volume numbers in hectoliters vs elasticity might produce large products. **Mitigation:** - We will scale units if necessary (maybe use thousands of HL as unit to avoid huge values). But given volumes ~10^7 HL and elasticity coefficients up to ~1, the linear terms might be ~10^7 in magnitude, which is okay for solvers. - We'll also avoid dividing by tiny base prices; if any SKU has a very low base price, ΔP/base_price in volume formula might blow up. But in our data, prices are in the thousands of COP, so fine. - We'll add a tiny epsilon in constraints where needed to avoid equality conflicts. For example, if base MACO = new MACO allowed, we might put new MACO >= base MACO * 1.001 to strictly improve, or just >= with solver tolerance. Solvers usually have a tolerance (like 1e-6) for satisfying constraints, so we rely on that.

Risk 6: User Misuse or Misinterpretation. The user might input unrealistic scenarios or misread outputs: - They might think 5% PINC guarantee 5% NR increase (not exactly, because volume drop can offset some). - Or they might not realize the constraints binding. **Mitigation:** - We include **guidance in the UI/documentation** (like tooltips: "High PINC might result in hitting volume constraints, limiting actual profit gain"). - We will handle edge cases: e.g. if user sets min bound > max bound (we'll validate and swap or error). - If user sets PINC slightly above 6 accidentally (our slider will cap at 6 but if they input text, we'll clamp it). - We'll make outputs clear: highlight that these are projections based on elasticity model, actual results may vary. - If any constraint is exactly at the limit (like volume dropped

exactly 1.0%), we might highlight it in orange to say “constraint active”. This informs user that pushing PINC more would break it. If all constraints are well within, we highlight slack.

Risk 7: Data Confidentiality and Compliance. Since this is hackathon, likely not an issue, but in a real scenario, pricing optimization output is sensitive. We must ensure the app doesn’t inadvertently expose data outside authorized use. **Mitigation:** - For hackathon, all data is internal. But if deploying, we’d implement authentication for the web app or integrate with internal systems. We mention this in future considerations if needed.

Risk 8: Timeline and Complexity Overrun. Building both a Streamlit and Next.js in hackathon timeframe is ambitious. We risk not fully polishing one. **Mitigation:** - We will prioritize having the **Streamlit version fully functional** as it covers core requirements and easier integration. The Next.js is a parallel effort that we plan but if time is short, we will perhaps provide a prototype or at least the design plan (which we have done in writing). The dual approach is in our plan, but delivery will ensure at least one robust UI (Streamlit) to avoid risking having two half-done ones. - We will do incremental integration: get optimization working in a notebook first, then integrate to Streamlit, then work on nicer UI elements, etc., to always have a running product.

Risk 9: Post-Optimization Manual Adjustments. Perhaps the optimal solution might violate some unwritten business preference (like maybe they never decrease price of a Premium SKU even if elasticity suggests it, or they want round pricing). We have not explicitly put rounding to nearest 50 (we did multiples of 50, which is fine), but what about final digit conventions or psychological price points? If needed, one might want to adjust a recommended price from 2,120 to 2,100 for simplicity. **Mitigation:** - Our tool can’t inherently know that (unless we add those as constraints like price endings). But we can address it by allowing the user to override specific SKUs and re-run. If a user doesn’t like 2120, they can set that SKU’s price bound to 2100–2100 and run again to re-optimize others around that fixed price. This manual override capability (perhaps via the UI advanced options) mitigates cases where the optimal solution might not be exactly what they implement – the tool can accommodate user adjustments and show the new outcome. - We’ll document this workflow (like “if you need a specific SKU price, set its bounds equal to fix it, then optimize remaining SKUs”).

By identifying these risks early, we can either incorporate solutions now or have a clear plan if they arise. During development, we will keep an eye on these issues (especially solver feasibility and performance). The combination of careful constraint handling, user input validation, thorough testing, and UI feedback will mitigate most of these.

Finally, we’ll document any remaining limitations. For instance, we will state: “This model assumes competitor prices static; a real scenario where competitors react would require an extended game-theoretic model (future work).” Acknowledging such limitations ensures transparency, and our mitigations (like not pushing to edges or allowing manual tweaks) ensure the tool remains practical and trustworthy.

Future Work & Enhancements

While our solution will meet the requirements of the Round 2 RGM problem, there are several areas for future enhancement and refinement. Given more time, data, or scope for development, we would consider the following improvements:

1. Refine Demand Modeling (Beyond Linear Elasticity): Our current model uses a linear approximation of volume response. In reality, price-volume relationships can be nonlinear (e.g.,

diminishing returns to price cuts, thresholds for consumer behavior). A future version could implement a **nonlinear demand model**, such as a constant elasticity (log-log) model or even a machine learning model that predicts volume given prices. This could be integrated into the optimization via piecewise linear approximations or using solvers that handle nonlinear programs. We could also incorporate **stochastic elasticities** – recognizing that elasticity estimates have confidence intervals, we might optimize for robust outcomes (e.g., guarantee improvement even if true elasticity is off by some margin). This would make the recommendations more resilient.

2. Competitor Price Reaction Scenarios: In the current model, competitors are static. A valuable extension is to incorporate scenarios where competitors might also change prices (which is realistic in RGM planning). We could allow the user to input assumptions about competitor price changes (e.g., "Competitor X will also take +3% on average") and feed that into the industry volume and cross-elasticity effects. Ultimately, we could model a simple **game theory** aspect: for example, optimize ABI's prices under worst-case competitor responses or find a Nash equilibrium if we had competitor elasticity matrix as well. This is complex but could be approximated by iterating the optimization with competitor moves. In a simpler form, just letting the user toggle competitor price increase vs hold could be an option.

3. Additional Business Constraints: - **Pricing Guidelines & Psychology:** Ensure prices end in certain digits (e.g., "99" endings or nearest 50 is already done, but perhaps nearest 100 for higher prices). We could incorporate rules like "don't price above certain psychological thresholds unless needed". - **Revenue Bucket Targets:** Perhaps management might have targets like "ensure at least X% revenue growth from premium segment" or "don't reduce any SKU's volume by more than Y%". We could add such constraints as needed. - **Inventory/Supply Constraints:** If supply is limited for certain SKUs, the optimizer might not want to boost their volume too much. We could integrate a constraint like Volume_new_i $\leq$ some capacity. With more data, e.g. inventory levels or production capacity, we could easily add that. - **Promotion & Calendar Integration:** Price changes often tie with promotional calendars. In future, we could incorporate the effect of promotional discounts or frequency. That might tie into Round 4 in some hackathons, but essentially optimizing not just base price but promo depth. This becomes a multi-period optimization.

4. Multi-Period and Long-Term Effects: The current model is static (one-shot price change). We could extend it to a multi-period simulation – for instance, plan a sequence of price increases (like 3% now, 3% in six months) and optimize that sequence for long-term MACO, considering maybe competitor catch-up or consumer adaptation. Additionally, connecting to **Brand Power (LTE)**: some price changes might erode brand equity long-term. If we had a model from the LTE task linking marketing and brand power, we could include a **term in the objective for brand health** or treat it as a constraint ("don't reduce power by more than …"). That integration would make the tool more holistic – balancing short-term profit and long-term brand equity.

5. Enhanced UI/UX: - **What-if Simulator:** Beyond optimization, allow users to manually tweak some prices and see projected outcomes instantly (a "sandbox mode"). For example, after optimization, a user might say "What if we also increased this Value SKU by an extra 50?" The UI could let them adjust that and quickly recalc volumes via the elasticity model (keeping others constant) to see impact. This gives more agency to users for exploration. - **Scenario Saving & Comparison:** The UI could allow saving multiple runs (e.g., Scenario A: PINC 3%, Scenario B: PINC 5% with different assumptions) and then show a comparison dashboard. This would be useful in strategy discussions to compare, say, a conservative vs aggressive pricing strategy side by side (volume, profit, share differences). - **Next.js Full Implementation:** Complete the transition to a fully featured Next.js front-end with proper authentication (for internal use), better styling, and integration with AB InBev's systems (e.g., pulling latest data directly from a database rather than Excel files). We would also ensure it's mobile-friendly if

needed (managers might view high-level results on tablet or phone). - **Gen AI Chatbot Integration:** As hinted in the hackathon guidelines [32] [33], we could add a **chatbot assistant** in the UI where a user can ask questions like "Why did the optimizer not use the full 6% PINC?" or "Which SKUs were most impacted?" The bot could be powered by an LLM fine-tuned on our results and methodology, giving explanations in human terms. For example, "It appears volume constraints prevented using the entire price increase. Most price increases were allocated to Premium SKUs due to their lower elasticity, maximizing profit." This would greatly improve interpretability. - **Localization:** If deploying in different markets, provide support for different currencies, languages (the UI could be translated to Spanish for Colombia team, etc.), and date/number formats.

6. Automated Data Update & Continuous Learning: Set up the tool to automatically update with new data. For instance, as more sales data comes in, re-run Round 1 elasticity estimation (perhaps make it a periodic process or even incorporate an online learning algorithm to update elasticities). The optimizer could then always use the latest elasticity matrix. We could containerize the elasticity model training from Round 1 and include it in the pipeline (with a toggle to retrain if needed). This way, the tool remains accurate over time. Additionally, we could monitor actual outcomes after implementing a price change and compare to our predictions, refining our elasticity estimates accordingly (closing the loop with real-world feedback).

7. Extend to Other RGM Levers: Real RGM decisions include not just pricing but also product mix, package innovation, etc. Our framework could be extended to, say, **optimize pack pricing and mix**: maybe suggest discontinuing a pack that's unprofitable or launching a new pack size at an optimal price point identified by elasticity gaps. This goes beyond current scope but the idea would be to find if a certain segment has a big elasticity, maybe an opportunity for a new SKU or a promotion. This becomes more of an advisory system using the same data.

8. More Granular Constraints – Micro-market & Channel: The current model is at national level (market = Colombia total). RGM might require ensuring certain regions or channels also meet targets. If data were available, we could run optimizations by region or channel or add constraints like "ensure volume in modern trade doesn't drop by more than X%" etc. Essentially, a multi-segment optimization which could be solved either by segmentation or adding dimensions to decision variables (increasing complexity significantly). In future, this could be addressed by running separate optimizations per segment or a hierarchical model.

9. UI Polishing & Collaboration Features: Add user accounts, ability to share a scenario with a colleague (so multiple users can view the same results), commentary fields where someone can annotate why they might override a recommendation, etc., making the tool more collaborative in a corporate environment.

10. Documentation & Training: Develop training materials and incorporate them into the app (like an interactive tutorial mode for new users). Also, integration of our documentation (MkDocs) into an internal wiki or help section in the app would be useful.

Many of these enhancements go beyond the hackathon's immediate needs, but they show a roadmap for evolving the solution into a comprehensive **Revenue Growth Management platform**. Our current implementation is a strong foundation: it can be trusted to give optimal solutions under defined conditions. These future improvements would increase its scope, accuracy, and user-friendliness, ultimately driving more value.

We prioritized building a solution that's **extensible**, so adding these features is feasible. For example, modular constraints allow adding a new constraint easily; a well-structured UI codebase can accommodate new components or pages. We have documented and parameterized key parts, meaning future developers (or ourselves) can adjust things like the elasticity approach or constraints without starting from scratch.

In summary, while the present solution tackles the immediate challenge of price optimization with constraints, the above future work items could transform it into a richer decision-support system. We will outline these in our project report to demonstrate forward-thinking to judges and stakeholders, showing that we understand the problem domain deeply and know how to enhance the tool as needs evolve.

Conclusion

In this deep research and build plan, we have outlined a comprehensive solution for AB InBev's RGM Pricing Optimizer, addressing both the technical modeling and the practical implementation aspects. Our approach leverages the Round 1 elasticity insights and applies them in a rigorous optimization framework to recommend profit-maximizing price changes at the SKU level, all while **strictly adhering to business constraints and preserving strategic price hierarchies**.

We began by clearly defining the **scope and success criteria** – maximizing MACO at the national level and ensuring no violations of volume, share, and pricing rules. We then detailed the **inputs and assumptions**, making use of all provided data (elasticities, sell-in/out, price list, etc.) and embedding domain knowledge (tax rates, cost inflation, typical PINC range) into our model. By doing so, we ensure the optimizer's calculations are grounded in reality, not abstract numbers.

The heart of our plan is the **Optimization Model**, where we translated business requirements into mathematical expressions. We formulated decision variables for price changes in discrete steps and built an objective function to increase profit. We thoroughly listed the **hard constraints** (industry volume, financial target, volume and share thresholds, PINC allocation, price step and bounds) [34] [3] and **soft constraints** (segment and pack price ladders) [4], and we described exactly how these will be implemented in the solver. By doing this, we've essentially written the "specification sheet" for the optimizer – it will not deviate from these rules, giving stakeholders confidence that all decisions are within agreed guardrails.

On the **solver strategy**, we chose a robust MILP approach using open-source tools, ensuring that the solution is **repeatable and optimal** rather than heuristic. We have planned for potential complexities (non-linearity) by linearizing and using an MIQP if needed. This shows our awareness of the computational side and guarantees that the tool can run quickly on a normal machine – a critical requirement for practical use.

From a user perspective, we designed an intuitive **UI/UX** with two pathways: an immediate Streamlit app for the hackathon deliverable and a vision for a polished Next.js interface for longer-term use. The UI focuses on simplicity (just input PINC and go) and clarity of outputs (detailed tables and charts), fulfilling the deliverable of a dynamic visualization of KPIs and price architecture [35]. We also factored in integration considerations (BrewVision, single-command deployment) and ensured our solution could be easily executed and evaluated by the judges.

We provided a realistic **demo script** that walks through using the tool, demonstrating that we've thought through the end-to-end experience and potential outcomes. This scenario highlighted how the

model reacts at the boundaries (volume drop hitting 1%, etc.), which again reinforces that we understand the interplay of constraints.

The **repository structure and documentation plans** emphasize maintainability and clarity. By structuring our code modularly and documenting it well (with citations to the rule source [34] [3] for traceability), we make it easier for others to trust and extend our work. Our testing strategy further underlines reliability – we are not just throwing algorithms together; we are validating each piece to ensure the final output is correct and respects the requirements exactly. This disciplined approach to quality will be evident to the AI code checker and judges in aspects like consistent style, meaningful tests, and lack of hacky code.

In concluding, we reiterate how our solution meets the **success criteria**: - **Maximized MACO:** The optimizer finds the best profit outcome given the constraints, as seen in our demo where MACO improved and we cited the formula used [12]. This aligns with the core goal of "maximizing Margin of Contribution" [36]. - **Constraint Compliance:** Every hard constraint from the Round 2 instructions is directly built into the model (we cited each from the PDF to leave no doubt) – volume change ≤5%/≥–1% [3], share drop ≤0.5% [10], PINC 0–6% [10], etc. The solution will never propose an action outside these limits. - **Business Logic Preservation:** Soft constraints like the segment and size hierarchy are maintained, so the new pricing is **strategically sound** (Value brands remain cheapest, premium brands priciest per HL [37]). This ensures the optimized plan would be acceptable to marketing and brand managers, not just a theoretical maximum. - **Usability:** The front-end provides actionable outputs – a clear list of new prices and their impact on volume, revenue, and profit, plus visual confirmation via charts. This empowers revenue managers to make informed decisions and easily communicate them (the summary table and charts can be plugged into internal reports or presentations). - **Integrability:** We designed the deployment such that AB InBev's team can incorporate this tool into their existing workflow with minimal fuss (no special environment needed beyond standard Python/Node, and data is read from provided files which can later be linked to databases).

We also indicated how this solution can be a stepping stone for broader capabilities (tying into long-term brand equity, multi-scenario planning, etc.). This forward-looking perspective shows that our solution is not a one-off hack, but a foundation that AB InBev could build on for continuous RGM optimization.

In conclusion, our plan provides a **detailed, implementation-ready roadmap** for delivering a Revenue Growth Management Pricing Optimizer that is both **technically sound** and **business-aligned**. By following this plan, we will produce a tool that delivers tangible value: optimal pricing recommendations that boost profitability while safeguarding market share and brand positioning. We've combined deep research (ensuring every number and constraint is justified by given data [34] [3]) with practical engineering (ensuring the solution can actually be used and trusted). We are confident that this solution will meet and exceed the Round 2 expectations, providing AB InBev a powerful new capability in their BrewVision toolkit and showcasing a successful integration of AI/data science into strategic decision-making.

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [31] [32] [33] [34] [35] [36] [37] Hackathon_2025_ProblemStatement R1.pdf
file://file-HWg9FiaAA37xJ3sWUAXi4c

[27] [28] [29] [30] Round 2 Instructions.pdf
file://file-DafmUqLkdgeuZntRg8Ey7X