Open-Source Optimization Framework for Fantasy Premier League: Data Engineering, Multi-Paradigm Modeling, and Backtesting Tools

Siu Hang IP

Assignment 4 - Final Report

December 5, 2025

Contents

1 Introduction

1.1 Project Overview

The motivation for this project stems from a personal interest in Fantasy Premier League (FPL) and observing fellow players make suboptimal decisions through manual, intuition-based methods. FPL is a popular online fantasy sports game where participants manage virtual football teams by selecting real Premier League players within a budget constraint, earning points based on actual match performances. Each gameweek presents managers with strategic decisions: which players to transfer in or out, who to captain for double points, and how to arrange starting lineups subject to formation rules. Regardless of whether managers rely on intuition, advanced machine learning predictions, or expert analysis, the team selection problem remains a completely separable discrete optimization problem that can be solved mathematically. Fantasy Premier League team management presents a discrete optimization problem combining multiple constraint categories and sequential decision-making. The core challenge involves selecting 15 players from the Premier League players within a £100 million budget, subject to positional quotas, formation rules for the starting XI, and club affiliation limits. Each gameweek introduces dynamic decisions: executing transfers, selecting a captain for double points, and arranging lineups subject to automatic substitution rules that replace non-playing starters with bench players.

This project seeks to formally define the mathematical structure of FPL team management and demonstrate how optimization techniques can systematically solve this discrete decision problem. By developing open-source optimization and backtesting tools, this work aims to create a valuable resource for students and enthusiasts to both learn about optimization in a relatable domain and apply these techniques to improve their actual FPL team management, without requiring deep, domain-specific expertise. For this study, data from three independent sources was integrated and cleaned, with the process documented in Python notebooks provided for transparency and reproducibility. The project delivers two primary tools subject to this integrated dataset: a team optimizer implementing both Mixed Integer Linear Programming (MILP) and Constraint Programming (CP) approaches with multiple solver backends, and a comprehensive backtesting infrastructure that validates decisions against actual 2024-25 season outcomes. The cleaned, integrated dataset spanning all 38 gameweeks of the 2024-25 season is also provided to support further research and experimentation.

2 Related Work

2.1 Literature Review

The literature on fantasy sports optimization has expanded alongside data accessibility and computational advancements, frequently conceptualizing team selection as constrained optimization problems.

Ramezani and Dinh (2025) [11] develop a data-driven MILP framework for FPL team selection, optimizing the full 15-player squad—including starting 11, bench, and captain—to maximize expected points under budget, position, and club constraints. Their hybrid scoring integrates historical data with linear regression predictions, evaluated on the 2023/24 season via Monte Carlo simulations and ARIMA forecasting, extending to multi-week transfer planning. This

informs the current project's MILP formulation, especially squad and captaincy with bench integration. However, it omits automatic substitutions and transfer mechanics—such as free transfer accumulation, four-point penalties, and 50% profit locks—deferring them to future work. These gaps are filled here via hierarchical pre-GW1 and single-stage post-GW1 models optimizing transfers with lineups, augmented by CP for efficiency comparison.

Becker and Sun (2016) [1] propose an analytical approach for fantasy football draft and lineup management, utilizing mixed-integer programming for NFL fantasy drafts and weekly lineups to maximize projected wins. Their robust modeling of player projection uncertainties from historical and expert data, combined with positional constraints, informs key design choices in the current work. Specifically, their treatment of uncertainty without explicit stochastic variables encouraged the adoption of conditional expected scores as inputs, accounting for factors like player availability. Additionally, their hierarchical approach to lineup optimization—applied in the draft phase to ensure positional depth for starters plus injury buffers before weekly starter selections—inspired similar prioritization in this project's formulation, adapted to FPL's unique rules such as club limits and substitution mechanics.

Groos (2025) [5] introduces OpenFPL, an open-source forecasting method that rivals state-of-the-art Fantasy Premier League services. This ensemble model combines XGBoost and Random Forest, trained on public data from the 2020-21 to 2023-24 seasons, with prospective evaluation on 2024-25. It forecasts points over one- to three-gameweek horizons, excelling in high-return player predictions. This work exemplifies the prediction-focused research strand in FPL analytics, which complements the current project's optimization focus. By treating predictions as external inputs, this project concentrates on the mechanics of team selection, transfers, and substitutions, making the work more accessible as an educational resource and allowing practitioners to integrate predictions from various sources, including statistical models like OpenFPL or expert judgment.

2.2 Data Provision and Infrastructure

The FPL-Elo-Insights GitHub repository [9] offers a comprehensive dataset for the 2024-2025 FPL season, fusing official FPL API data with detailed match statistics, dynamic team Elo ratings, and coverage of cups, friendlies, and European competitions. Organized into directories with high-level summaries and granular gameweek snapshots in CSV format, it aligns data by official FPL IDs, enabling deep analysis of player and team performance. This resource was utilized in the current study for integrating real-season data, addressing challenges in schema inconsistencies and temporal alignment as noted in the implementation.

The Fantasy-Premier-League GitHub repository [14] provides CSV datasets for the 2024-25 FPL season, including player statistics, gameweek-specific data, team information, and season histories. It aggregates official FPL data into accessible formats like merged gameweek files, facilitating historical analysis and predictions. Employed in this project as a primary data source, it supported validation against actual outcomes, though updates ceased after the 2024-25 season.

The fplcache GitHub repository [12] maintains a daily cache of Fantasy Premier League bootstrap data from the official API, including player and team details in LZMA-compressed JSON format. It features scripts for caching, diffing files, and extracting player data, aiding in efficient data access and cross-referencing without repeated API calls. This served

as an honorable mention in the current study for understanding FPL API structures and supporting data integration efforts.

3 Design

3.1 Incorporating Feedback from Assignment 4A

The design evolved significantly from the initial Assignment 4A proposal, drawing on three key sources of feedback and inspiration. First, inspired by David Bergman's practical application of optimization to sports analytics, this project integrates three independent datasets, covering the full 2024-2025 FPL season across 38 gameweeks with actual player statistics and prices. This choice ensures the framework tackles real-world challenges and validates outcomes against historical results.

Second, to enhance realism as suggested, the system fully models FPL mechanics. Substitution logic follows automatic rules, accounting for player availability, positional needs, and formation limits. Transfers include weekly accumulation of one free transfer, with four-point penalties for extras. Trading distinguishes buying from selling prices, enforcing the 50% profit lock and enabling precise budget tracking. All elements respect data availability and quality from the sources.

Third, per recommendations for thorough evaluation, the system adopts two modeling paradigms: Constraint Programming (CP) and Mixed Integer Linear Programming (MILP). It supports multiple backends, such as CBC [3] (PuLP default [7]), GLPK [6], and SCIP [4] for MILP, plus MiniZinc [8] solvers including CP-SAT [10], Chuffed [2], and Gecode [13] for CP. This setup allows direct comparison of approaches and solver efficiency.

Additionally, the design simplifies scoring by using conditional expected points, which embed availability likelihood into a single metric per player. This avoids separate stochastic variables for no-shows and probabilities, streamlining the objective. The proposed hybrid MILP/CP decomposition was dropped, as standalone MILP solvers proved efficient, boosting interpretability, accessibility, and simplicity.

3.2 Scope

This FPL optimization system adopts a focused scope to balance feasibility and utility, with these core constraints and elements. The model optimizes for a single upcoming gameweek, using current state and next-week predictions only. It ignores prior or multi-week data, cutting complexity without sacrificing relevance. Objectives remain linear, matching FPL scoring and solver capabilities. Non-linear options, like variance-minimizing portfolios, are omitted due to unavailable prediction variances in public data. Predictions serve as external inputs: conditional expected points drive team selection. This decouples optimization from forecasting, supporting diverse sources like models or expert input.

Core FPL elements receive full coverage. Single-gameweek optimization handles squad, transfers, captain, and lineup, guided by next-week predictions. Rule enforcement includes budgets, squad quotas, formations, club limits, and bench/-substitution handling. Transfer mechanics cover weekly free transfers, four-point penalties for extras, 50% profit locks on sales, and per-player price tracking for budgets. Automatic substitutions replace non-starters from bench, validating

formations throughout. Captaincy awards double points for captain (vice fallback if absent), assigned post-optimization by expected points. Data handling merges three sources, resolves conflicts via validation, deduplicates fixtures, and normalizes participation. Backtesting simulates the 2024-25 season over 38 weeks, with per-week reports, cumulative metrics, transfer analysis, and rule checks.

For focus and manageability, certain aspects are omitted. Multi-week planning, including rolling horizons for prices, fixtures, or squad value, is not pursued. Chip strategies (Wildcard, Bench Boost, Triple Captain, Free Hit) are excluded, as they demand multi-period analysis. Prediction development receives no attention: no training or evaluation of models; inputs are external. Risk or non-linear goals are absent: no variance penalties, CVaR, or utility curves for manager preferences.

This scope yields a robust, validated framework, sidestepping added complexity from broader features.

3.3 Design Principles

Two principles shaped the design. The system excels with precise predictions, prioritizing starting-11 optimization. Scores derive from starters, assuming accurate inputs select reliable players and minimize no-shows. This favors peak performance over uncertainty hedges via intricate benches.

Installation and operation span platforms effortlessly. PuLP (MILP) offers pure-Python ease with versatile backends; MiniZinc (CP) provides a solid Python API. Both avoid platform quirks, broadening access for users and researchers.

3.4 Technical Challenges During Implementation

Implementation revealed four primary technical challenges, each addressed through targeted strategies to ensure robustness.

Data integration across three independent sources demanded rigorous reconciliation. Schema inconsistencies—varying naming conventions, ID systems, and structures—required fuzzy matching and manual verification for accurate player identification. Temporal alignment ensured correspondence across gameweeks, while completeness checks prioritized primary sources supplemented by others for full coverage.

Data quality issues necessitated thorough cleaning. Missing values in statistics and prices were imputed or handled judiciously. Duplicates, often from mid-season transfers, underwent deduplication that preserved timelines. Inconsistent formats, such as price notations ("£10.5M" versus 105), were standardized. Validation confirmed essentials like 15-player squads and compliant formations prior to optimization.

Developing the Constraint Programming model in MiniZinc required adaptation to its declarative paradigm, distinct from Python's imperative style. The Python API's nuances in data passing and retrieval demanded precise type conversions. Set operations for constraints like club affiliations proved tricky, especially with intertwined membership and cardinality rules. Limited debugging tools hindered tracing unsatisfiable constraints or errors.

CP solvers faced performance hurdles, rendering them less viable. Timeouts frequently exceeded minutes for even modest instances. Scalability faltered with growing player pools or constraints. Solution times varied unpredictably across cases.

By contrast, MILP solvers like CBC and SCIP resolved large instances in seconds, affirming their reliability for this domain.

3.5 Architecture, Algorithms, and Methodologies

3.5.1 Notation and Definitions

Sets, Parameters and Variables

- P: Set of all players.
- C: Set of clubs.
- $G \subset P$: Goalkeepers (GK).
- $D \subset P$: Defenders (DEF).
- $M \subset P$: Midfielders (MID).
- $F \subset P$: Forwards (FWD).
- $S \subset P$: Set of starting XI players.
- $B \subset P$: Set of bench players.
- $Q = S \oplus B \subset P$: Set of all squad players.
- $p_i \in \mathbb{R}$: Cost of player $i \in P$.
- $u_i \in \{0, 1\}$: 1 if player $i \in P$ is unavailable for the upcoming gameweek, 0 if available.
- $y_i^0 \in \{0, 1\}$: 1 if player i is in current squad (known state, for Post-GW1).
- $B_{\text{bank}} \in \mathbb{R}^+ \cup \{0\}$: Cash in bank (known state, for Post-GW1).
- $p_{\text{buy},i}^0 \in \mathbb{R}^+ \cup \{0\}$: Purchase price paid for player $i \in P$
- $sp_i \in \mathbb{R}^+$: Selling price for player $i \in P$.
- $f \in \{0, 1, 2, 3, 4, 5\}$: Free transfers (1 + banked, ≤ 5).
- expected_points$_i$: Conditional expected points for player $i \in P$ in upcoming gameweek.
- $x_i \in \{0, 1\}$: 1 if player $i \in P$ is in the starting XI.
- $c_i \in \{0, 1\}$: 1 if player $i \in P$ is captain.
- $y_i \in \{0, 1\}$: 1 if player $i \in P$ is in the squad.
- $t_i \in \{0, 1\}$: 1 if buy player i (transfer in).

- $s_i \in \{0, 1\}$: 1 if sell player i (transfer out).
- $e \in \{0, 1, 2, \ldots, 15\}$: Number of extra transfers beyond f (penalty).

3.5.2 Pre-GW1 Hierarchical Optimization

The system adopts a two-stage hierarchical optimization process for squad selection. As team scores derive exclusively from the starting 11 players (unless substitutions), this method prioritizes lineup composition to maximize potential performance. The bench is then optimized with residual resources to ensure adequate contingency coverage.

Step 1: Optimize Starters and Captain

Decision variables: $y_i, x_i, c_i \in \{0, 1\}$ for all $i \in P$.

Solve the following optimization problem:

$$\max \quad \sum_{i \in P} \text{expected\_points}_i \cdot (x_i + c_i) \tag{1}$$

subject to constraints:

$$\sum_{i \in P} x_i = 11, \quad \text{(lineup size)}$$

$$\sum_{i \in G} x_i = 1, \quad \text{(1 starting GK)}$$

$$3 \leq \sum_{i \in D} x_i \leq 5, \quad \text{(3-5 starting DEF)}$$

$$2 \leq \sum_{i \in M} x_i \leq 5, \quad \text{(2-5 starting MID)}$$

$$1 \leq \sum_{i \in F} x_i \leq 3, \quad \text{(1-3 starting FWD)}$$

$$\sum_{i \in P} c_i = 1, \quad \text{(exactly 1 captain)}$$

$$c_i \leq x_i \quad \forall i \in P, \quad \text{(captain must be starter)}$$

$$u_i + x_i \leq 1 \quad \forall i \in P, \quad \text{(unavailable players cannot be starter)}$$

$$\sum_{i \in P} y_i = 15, \quad \sum_{i \in G} y_i = 2, \quad \sum_{i \in D} y_i = 5,$$

$$\sum_{i \in M} y_i = 5, \quad \sum_{i \in F} y_i = 3, \quad \text{(squad quotas)}$$

$$\sum_{i \in P} p_i y_i \leq 100, \quad \sum_{i \in P_c} y_i \leq 3 \quad \forall c \in C, \quad \text{(budget and club limits)}$$

$$x_i \leq y_i \quad \forall i \in P \quad \text{(starters must be in squad)}$$

Denote the optimal solution from Step 1 as x_i^*, y_i^*, c_i^*.

Post-hoc: Vice-Captain Assignment The vice-captain is assigned post-hoc following the primary optimization, rather than integrated into the model. This simplifies the formulation, as the vice-captain does not influence the objective function.

Let $S = \{i \in P \mid x_i^* = 1\}$ denote the selected starting players. After Step 1, exclude the captain from S and designate the vice-captain as the remaining starter with the highest expected points. This secures the optimal backup for captain no-shows without affecting the initial optimization.

Step 2: Optimize Bench Composition

Decision variables: $y_j \in \{0, 1\}$ for all $j \in P$.

State variable: $x_i^* \in \{0, 1\}$ from Step 1 for all $i \in P$.

Solve the following optimization problem:

$$\max \quad \sum_{j \in P} \text{expected\_points}_j \cdot y_j \tag{2}$$

subject to:

$$\sum_{i \in P} y_i = 15 \quad \text{(total squad size)}$$

$$\sum_{i \in G} y_i = 2, \quad \sum_{i \in D} y_i = 5, \quad \sum_{i \in M} y_i = 5, \quad \sum_{i \in F} y_i = 3 \quad \text{(position quotas)}$$

$$\sum_{i \in P} p_i y_i \leq 100, \quad \sum_{i \in P_c} y_i \leq 3 \quad \forall c \in C \quad \text{(budget and club limits)}$$

$$y_i = x_i^* = 1 \quad \forall i \in S \quad \text{(fix starters from Step 1)}$$

Denote the optimal solution from step 2 as y_i^{**}.

Post-hoc: Bench Position Assignment Bench ordering, like vice-captain selection, occurs post-hoc, as it influences the objective only during substitutions. The process prioritizes high-expected-point players for early entry, per FPL substitution rules:

1. Place the backup goalkeeper (non-starting GK) in position 1.
2. Rank the three benched outfield players by descending expected points.
3. Assign the highest-ranked to position 2 (first substitute), the middle to position 3, and the lowest to position 4 (last substitute).

This allocates one player per position, with the goalkeeper in 1 and outfielders in 2–4.

Output for Pre-GW1 The final output from the Pre-GW1 formulation consists of:

- Squad composition: $y_i^{**} \in \{0,1\}$ for all $i \in P$ (15 players).
- Starting lineup: $x_i^* \in \{0,1\}$ for all $i \in P$ (11 players).
- Captain: $c_i^* \in \{0,1\}$ for all $i \in P$ (1 player).
- Vice-captain: assigned post-hoc as per above.
- Bench positions: assigned post-hoc as per above.

Initial Condition for State Variables After completing GW1 optimization, initialize the state variables for GW2:

$$y_i^0 \leftarrow y_i^{**} \quad \forall i \in P \quad \text{(GW1 squad becomes initial squad)}$$

$$B_{\text{bank}} = 100 - \sum_{i \in P} p_i y_i^{**} \quad \text{(remaining cash)}$$

$$f = 2 \quad \text{(1 free transfer per week, starts with 1 banked)}$$

$$p_{\text{buy},i}^0 = \begin{cases} p_i & \text{if } y_i^{**} = 1 \quad \text{(GW1 squad members)} \\ 0 & \text{if } y_i^{**} = 0 \quad \text{(all other players)} \end{cases} \quad \forall i \in P$$

These state variables form the complete initial state for the Post-GW1 formulation.

3.5.3 Post-GW1 Single-Stage Optimization

Post-GW1, the system employs single-stage optimization to select the full 15-player squad simultaneously, omitting separate bench re-optimization. This prevents transfers—free or penalized—from being expended on marginal bench improvements alone.

Selling Price Calculation The current squad y_i^0 is known. For each player $i \in P$ in the current squad ($y_i^0 = 1$), the selling price sp_i implements FPL's 50% profit lock mechanism:

$$sp_i = p^0_{\text{buy},i} + 0.5 \cdot \max(0, p_i - p^0_{\text{buy},i})$$

rounded up to the nearest 0.1m if there is profit ($p_i > p^0_{\text{buy},i}$); otherwise, $sp_i = p_i$ if there is a loss ($p_i \leq p^0_{\text{buy},i}$).
This formulation ensures that players sold at a gain contribute only half their price increase, with the profit locked at 50%, while players sold at a loss contribute their full current price, realizing losses completely.

Budget Dynamics Only purchases of new players incur costs; retained players impose none, as they are already owned. The budget constraint ensures non-negative cash post-transfers.
From the state update:

$$B^{\text{new}}_{\text{bank}} = B_{\text{bank}} + \sum_{i \in P} sp_i s_i - \sum_{j \in P} p_j t_j$$

Requiring $B^{\text{new}}_{\text{bank}} \geq 0$ gives:

$$B_{\text{bank}} + \sum_{i \in P} sp_i s_i - \sum_{j \in P} p_j t_j \geq 0$$

Rearranging yields the budget constraint:

$$\sum_{j \in P} p_j t_j \leq B_{\text{bank}} + \sum_{i \in P} sp_i s_i$$

where p_j is the current market price for new purchases and sp_i is the locked selling price (with 50% profit lock) for any sold players.

Step 1: Optimize Starters, Captain, and Transfers

Decision variables: $x_i, y_i, c_i, t_i, s_i \in \{0, 1\}$ for all $i \in P$, and $e \in \{0, 1, 2, \ldots, 15\}$.

State Variables:

- $y_i^0 \in \{0, 1\}$: Current squad composition (15 players)
- $B_{\text{bank}} \in \mathbb{R}^+$: Cash in bank (remaining budget)
- $p^0_{\text{buy},i} \in \mathbb{R}^+ \cup \{0\}$: Purchase price for player $i \in P$ (0 if not in squad)
- $f \in \{0, 1, 2, 3, 4, 5\}$: Free transfers available (typically $f = 2$ for GW2)

Solve the following optimization problem:

$$\max \quad \sum_{i \in P} \text{expected\_points}_i \cdot (x_i + c_i) - 4e \tag{3}$$

subject to constraints:

$$\sum_{i \in P} x_i = 11, \quad \text{(lineup size)}$$

$$\sum_{i \in G} x_i = 1, \quad \text{(1 starting GK)}$$

$$3 \leq \sum_{i \in D} x_i \leq 5, \quad \text{(3-5 starting DEF)}$$

$$2 \leq \sum_{i \in M} x_i \leq 5, \quad \text{(2-5 starting MID)}$$

$$1 \leq \sum_{i \in F} x_i \leq 3, \quad \text{(1-3 starting FWD)}$$

$$\sum_{i \in P} c_i = 1, \quad \text{(exactly 1 captain)}$$

$$c_i \leq x_i \quad \forall i \in P, \quad \text{(captain must be starter)}$$

$$u_i + x_i \leq 1 \quad \forall i \in P, \quad \text{(unavailable players cannot start)}$$

$$\sum_{i \in P} y_i = 15, \quad \sum_{i \in G} y_i = 2, \quad \sum_{i \in D} y_i = 5,$$

$$\sum_{i \in M} y_i = 5, \quad \sum_{i \in F} y_i = 3, \quad \text{(squad quotas)}$$

$$\sum_{j \in P} p_j t_j \leq B_{\text{bank}} + \sum_{i \in P} sp_i s_i, \quad \sum_{i \in P_c} y_i \leq 3 \quad \forall c \in C, \quad \text{(budget and club limits)}$$

$$x_i \leq y_i \quad \forall i \in P \quad \text{(starters must be in squad)}$$

$$y_i = y_i^0 + t_i - s_i \quad \forall i \in P \quad \text{(squad update)}$$

$$t_i + s_i \leq 1 \quad \forall i \in P \quad \text{(no simultaneous buy/sell)}$$

$$t_i \leq 1 - y_i^0 \quad \forall i \in P \quad \text{(buy only non-squad)}$$

$$s_i \leq y_i^0 \quad \forall i \in P \quad \text{(sell only current squad)}$$

$$e \geq \sum_{i \in P} t_i - f \quad \text{(extra transfers)}$$

$$\sum_{i \in P} t_i \leq 15 \quad \text{(logical upper bound on transfers)}$$

Denote the optimal solution from Step 1 as $x_i^*, y_i^*, c_i^*, t_i^*, s_i^*, e^*$.

Post-hoc: Vice-Captain and Bench Assignments Vice-captain and bench position assignments are performed post-hoc following the same procedures as in Pre-GW1.

Output for Post-GW1 The final output from the Post-GW1 formulation consists of:

- Squad composition: $y_i^* \in \{0,1\}$ for all $i \in P$ (15 players, from Step 1).
- Starting lineup: $x_i^* \in \{0,1\}$ for all $i \in P$ (11 players, from Step 1).
- Captain: $c_i^* \in \{0,1\}$ for all $i \in P$ (1 player, from Step 1).
- Vice-captain and Bench positions: assigned post-hoc.
- Transfers made: $t_i^*, s_i^* \in \{0,1\}$ for all $i \in P$ (from Step 1).
- Extra transfers penalty: e^* (from Step 1).

State Update for Next Gameweek After completing Step 1, update the state variables to carry forward to the next gameweek using per-player cash calculations:

Cash Update:

Compute the new bank balance based on actual selling proceeds and buying costs:

$$B_{\text{bank}}^{\text{new}} = B_{\text{bank}} + \sum_{i \in P} sp_i s_i^* - \sum_{j \in P} p_j t_j^* \tag{4}$$

where: $\sum_{i \in P} sp_i s_i^*$: Cash from selling players (using locked selling prices),

$\sum_{j \in P} p_j t_j^*$: Cost of buying players (at current market prices).

Purchase Price State Update:

Update the purchase price history for all players:

$$\forall i \in P: \quad \begin{cases} p_{\text{buy},i}^0 \leftarrow 0 & \text{if } s_i^* = 1 \quad \text{(sold players - clear history)} \\ p_{\text{buy},i}^0 \leftarrow p_i & \text{if } t_i^* = 1 \quad \text{(bought players - record purchase)} \\ p_{\text{buy},i}^0 \text{ unchanged} & \text{otherwise} \quad \text{(retained/non-squad players)} \end{cases} \tag{5}$$

Squad State Update:

Update the state variables for the next gameweek:

- $y_i^0 \leftarrow y_i^*$ for all $i \in P$ (new squad becomes current squad)
- $B_{\text{bank}} \leftarrow B_{\text{bank}}^{\text{new}}$ (updated cash in bank from equation above)
- $p_{\text{buy},i}^0$ updated as specified (purchase price tracking)
- f: Free transfers for next gameweek (1 per week, bankable up to max 5)

The decomposition is solver-independent and serves as an algorithmic blueprint for efficient FPL optimization.

3.5.4 CP-Based Formulations with High-Level Set Abstraction

This subsection presents equivalent formulations using declarative set-based abstraction, as implemented in MiniZinc constraint programming models. The set-based abstraction maps directly to the binary formulations presented above.

Sets and Parameters Let $P = \{1, \ldots, n\}$ be the set of all players.

Let $C = \{1, \ldots, 20\}$ be the set of clubs.

Let $\mathcal{P} = \{\text{GK}, \text{DEF}, \text{MID}, \text{FWD}\}$ be the set of positions.

For each position $p \in \mathcal{P}$, let $P_p \subseteq P$ be the players available in position p.

For each club $c \in C$, let $P_c \subseteq P$ be the players in club c.

Let $e_i \in \mathbb{R}_{\geq 0}$ be the expected points for player $i \in P$.

Let $k_i \in \mathbb{R}_{\geq 0}$ be the cost for player $i \in P$.

Let $u_i \in \{0, 1\}$ indicate if player $i \in P$ is unavailable.

Let $U \in \mathbb{R}_{\geq 0}$ be a logical upper bound on the objective.

Let $L \in \mathbb{R}_{\leq 0}$ be a logical lower bound on the objective.

Decision Variables The following set-based decision variables are common to both Pre-GW1 and Post-GW1 formulations:

Let $Q \subseteq P$ be the squad set, with $|Q| = 15$.

Let $S \subseteq Q$ be the starters set, with $|S| = 11$.

Let $K \subseteq S$ be the captain set, with $|K| = 1$.

For each position $p \in \mathcal{P}$, let $Q_p \subseteq P_p$ be the squad partition for p.

For each position $p \in \mathcal{P}$, let $S_p \subseteq Q_p$ be the starters partition for p.

Pre-GW1 Optimization Step 1: Optimize Starters and Captain

Solve the following optimization problem:

$$\max_{Q,S,K,\{Q_p,S_p\}_{p\in\mathcal{P}}} \quad z = \sum_{i\in S} e_i + \sum_{i\in K} e_i \tag{6}$$

subject to:

$$0 \leq z \leq U,$$

Cardinalities:

$$|Q| = 15, \quad |S| = 11, \quad |K| = 1,$$

$$|Q_{\text{GK}}| = 2, \quad |Q_{\text{DEF}}| = 5, \quad |Q_{\text{MID}}| = 5, \quad |Q_{\text{FWD}}| = 3,$$

$$|S_{\text{GK}}| = 1, \quad 3 \le |S_{\text{DEF}}| \le 5, \quad 2 \le |S_{\text{MID}}| \le 5, \quad 1 \le |S_{\text{FWD}}| \le 3,$$

Partitions:

$$Q = \bigcup_{p \in \mathcal{P}} Q_p, \quad \forall p_1 \neq p_2 \in \mathcal{P} : \, Q_{p_1} \cap Q_{p_2} = \emptyset,$$

$$S = \bigcup_{p \in \mathcal{P}} S_p, \quad \forall p_1 \neq p_2 \in \mathcal{P} : \, S_{p_1} \cap S_{p_2} = \emptyset,$$

$$\forall p \in \mathcal{P} : \, Q_p \subseteq P_p, \, S_p \subseteq Q_p,$$

Budget:

$$\sum_{i \in Q} k_i \le 100,$$

Club Limits:

$$\forall c \in C : \, |Q \cap P_c| \le 3,$$

Availability:

$$\forall i \in P : \, (u_i = 1) \implies (i \notin S),$$

Subsets:

$$S \subseteq Q, \quad K \subseteq S.$$

Denote the optimal solution from Step 1 as $Q^*, S^*, K^*, \{Q_p^*, S_p^*\}_{p \in \mathcal{P}}$.

Step 2: Optimize Bench

Fix S^* and $\{S_p^* \mid p \in \mathcal{P}\}$ from Step 1's optimal solution. Then solve:

$$\max_{Q, \{Q_p\}_{p \in \mathcal{P}}} \quad z = \sum_{i \in Q} e_i \tag{7}$$

subject to:

$$0 \leq z \leq U,$$
$$S = S^*, \quad \forall p \in \mathcal{P} : S_p = S_p^*,$$

and all other constraints from Step 1.

Denote the optimal solution from Step 2 as $Q^{**}, \{Q_p^{**}\}_{p \in \mathcal{P}}$.

Post-hoc Assignments:

After solving steps 1 and 2, perform the same post-hoc assignments as described earlier (vice-captain and bench position assignments).

Post-GW1 Optimization with Transfers The key design choice for the CP formulation is using explicit binary variables t_i, s_i for transfers (rather than set differences) to avoid propagation issues, while maintaining high-level set abstractions for squad/starters.

Additional State Variables:

For Post-GW1, the following state variables from the previous gameweek are required:

Let $y_i^0 \in \{0, 1\}$ indicate if player $i \in P$ is in the current squad.

Let $sp_i \in \mathbb{R}_{\geq 0}$ be the selling price for player $i \in P$.

Let $B_{\text{bank}} \in \mathbb{R}_{\geq 0}$ be the cash in bank.

Let $f \in \{0, 1, 2, 3, 4, 5\}$ be the number of free transfers available.

Additional Decision Variables:

For each player $i \in P$, let $t_i \in \{0, 1\}$ indicate transfer in (buy player i).

For each player $i \in P$, let $s_i \in \{0, 1\}$ indicate transfer out (sell player i).

Let $e \in \{0, 1, \ldots, 15\}$ be the number of extra transfers beyond f.

For each player $i \in P$, let $y_i \in \{0, 1\}$ indicate if player i is in the new squad (channeled with set Q).

Optimization Formulation:

Solve the following optimization problem:

$$\max_{Q,S,K,\{Q_p,S_p\}_{p \in \mathcal{P}},\{y_i,t_i,s_i\}_{i \in P},e} \quad z = \sum_{i \in S} e_i + \sum_{i \in K} e_i - 40 \cdot e \tag{8}$$

subject to:

$$L \leq z \leq U \quad \text{(precalculated logical bound)},$$

Transfer Logic:

$$\forall i \in P:\ y_i = y_i^0 + t_i - s_i \quad \text{(squad update)},$$

$$\forall i \in P:\ t_i + s_i \le 1 \quad \text{(no buy and sell same player)},$$

$$\forall i \in P:\ y_i^0 + t_i \le 1 \quad \text{(no buy if owned)},$$

$$\forall i \in P:\ s_i \le y_i^0 \quad \text{(only sell owned)},$$

$$e \ge \sum_{i \in P} t_i - f \quad \text{(extra transfers penalty)},$$

$$\sum_{i \in P} t_i \le 15,$$

Budget with Transfers:

$$\sum_{i \in P} k_i \cdot t_i \le B_{\text{bank}} + \sum_{i \in P} sp_i \cdot s_i,$$

Channeling and Subsets:

$$\forall i \in P:\ (y_i = 1) \iff (i \in Q),$$

$$S \subseteq Q, \quad K \subseteq S,$$

Cardinalities:

$$\sum_{i \in P} y_i = 15, \quad |S| = 11, \quad |K| = 1,$$

$$|Q_{\text{GK}}| = 2, \quad |Q_{\text{DEF}}| = 5, \quad |Q_{\text{MID}}| = 5, \quad |Q_{\text{FWD}}| = 3,$$

$$|S_{\text{GK}}| = 1, \quad 3 \le |S_{\text{DEF}}| \le 5, \quad 2 \le |S_{\text{MID}}| \le 5, \quad 1 \le |S_{\text{FWD}}| \le 3,$$

Partitions:

$$Q = \bigcup_{p \in \mathcal{P}} Q_p, \quad \forall p_1 \ne p_2 \in \mathcal{P}:\ Q_{p_1} \cap Q_{p_2} = \emptyset,$$

$$S = \bigcup_{p \in \mathcal{P}} S_p, \quad \forall p_1 \ne p_2 \in \mathcal{P}:\ S_{p_1} \cap S_{p_2} = \emptyset,$$

$$\forall p \in \mathcal{P}:\ Q_p \subseteq P_p,\ S_p \subseteq Q_p,$$

Club Limits and Availability:

$$\forall c \in C : |Q \cap P_c| \leq 3,$$

$$\forall i \in P : u_i = 1 \implies i \notin S.$$

Post-hoc Assignments:

After solving the optimization, perform the same post-hoc assignments as described earlier (vice-captain and bench position assignments). State updates follow the procedure outlined in the Post-GW1 section above.

4 Implementation

The implementation of the FPL optimization system consisted of five major phases: (1) Data Integration, (2) Data Cleaning, (3) MILP Model Implementation, (4) CP MiniZinc Model Implementation, and (5) Backtesting and Validation. This section details the technical implementation of each phase, highlighting key challenges, solutions, and noteworthy design decisions.

4.1 Implementation Overview

The system was implemented primarily in Python 3.10+, leveraging several key technologies and libraries:

- **Data Processing**: pandas, numpy, csv for data manipulation and integration
- **Optimization Solvers**:
  - MILP: PuLP library with CBC, GLPK, and SCIP solvers
  - CP: MiniZinc with CP-SAT, Chuffed and Gecode
- **Development**: Jupyter notebooks for exploratory work, Python scripts for production code

4.2 Phase 1: Data Integration

High-quality input data forms the foundation of any data-driven optimization system. A primary challenge for this Fantasy Premier League (FPL) team management system was the lack of a single open dataset providing complete time-series player buying prices across the 2024-25 season. FPL prices fluctuate daily by up to £0.1 million due to transfer demand, but available datasets offer only gameweek snapshots without intra-week timing details.

Three complementary open-source sources were integrated for cross-validation. The OLB Dataset (olbauday/FPL-Elo-Insights) supplies player statistics, prices (`now_cost`), status, and expected points. The VAA Dataset (vaastav/Fantasy-Premier-League) delivers detailed performance metrics, including minutes played and starting appearances. Randdalf's fplcache caches official FPL API data as a third reference.

This process spanned Gameweeks 1 to 38, producing a unified dataset with 27,222 unique (player, gameweek) records. The subsequent discussion outlines the technical challenges encountered and solutions implemented.

4.2.1 OLB Dataset Processing

The OLB dataset comprised two primary files: `players.csv` for player metadata and `playerstats.csv` for per-gameweek statistics. Processing entailed three essential steps.

Step 1: ID-to-Name Mapping The initial structural challenge arose from `playerstats.csv` using numeric player IDs, incompatible with the VAA dataset's player names. Mapping required a validation-first strategy to link IDs to full names.

The following implementation was employed:

Listing 1: Player ID Mapping with Validation

```
# Create id-to-name mapping
players_df['full_name'] = (players_df['first_name'] + ' ␣' +
                           players_df['second_name'])
id_to_name = dict(zip(players_df['player_id'],
                      players_df['full_name']))

# Validate: ensure all playerstats IDs exist in players.csv
player_ids = set(players_df['player_id'].unique())
playerstats_ids = set(playerstats_df['id'].unique())

if player_ids >= playerstats_ids:  # Superset check
    playerstats_df['player_name'] = playerstats_df['id'].map(id_to_name)
    mapped_df = playerstats_df.drop('id', axis=1)
```

Superset verification confirmed that all 631 distinct IDs in `playerstats.csv` matched entries in `players.csv`, preserving integrity for merging.

Step 2: Column Filtering The mapped dataset included extraneous variables, so we retained only those vital to the optimization model: `player_name`, `status`, `chance_of_playing_next_round`, `chance_of_playing_this_round`, `now_cost`, `event_points` (gameweek-specific FPL points), `ep_next` and `ep_this` (expected points), and `gw`.

A key insight emerged regarding `total_points`: in the OLB dataset, it denotes cumulative points from GW1, contrasting with the VAA dataset's gameweek-specific scoring. This semantic distinction informed later validations.

Step 3: Duplicate Verification Prior to integration, we assessed data quality by scanning for duplicate (gameweek, player name) pairs, anomalies given the standard one-match-per-club schedule.

Validation revealed zero duplicates across the OLB dataset's 27,658 records, yielding a reliable foundation. This cleanliness facilitated resolution of duplicates later identified in the VAA dataset.

4.2.2 VAA Dataset Processing

The VAA dataset introduced greater complexities, comprising 38 separate CSV files (`gw1.csv` through `gw38.csv`), one per gameweek.

Step 1: Column Inconsistency Detection Initial merge attempts faltered owing to mid-season structural alterations. Files `gw22.csv` through `gw38.csv` incorporated seven extraneous columns absent from earlier files:

```
Extra columns: mng_clean_sheets, mng_draw, mng_goals_scored,
               mng_loss, mng_underdog_draw, mng_underdog_win,
               mng_win
```

This schema evolution likely arose from upstream source modifications. The resolution employed conditional ingestion: direct reading for GW1–21; for GW22–38, exclusion of specified columns via index-based filtering to restore uniformity.

Step 2: Validated File Merging All 38 files were merged with enforced column alignment. Early files (GW1–21) underwent standard ingestion; later files (GW22–38) first confirmed structural parity post-exclusion, then proceeded with filtering. A gameweek identifier column was appended to each row for temporal fidelity. This yielded 27,605 rows of consistent data across the season.

Step 3: Column Filtering Analogous to OLB processing, the merged VAA data was pruned to indispensable fields: `name`, `position`, `team`, `xP` (expected points), `minutes`, `starts` (starting lineup indicator), `total_points` (gameweek-specific points), `value` (player price), and `gw`.

Step 4: Duplicate Detection and Analysis In contrast to the OLB dataset, duplicate scrutiny identified 374 (gameweek, name) pairs, clustered in GW24 (112 pairs), GW25 (98 pairs), GW32 (82 pairs), and GW33 (82 pairs). Each instance duplicated precisely twice, with divergent content per row. Examination of the raw data pinpointed the cause: fixture rescheduling from weather or other factors prompted double matches for select clubs, as indicated by disparate kickoff times and opponent codes.

4.2.3 Cross-Dataset Integration and Validation

Following independent processing of both datasets, merging required careful handling of duplicate fixture records.

Step 1: Hypothesis Testing for Duplicate Validation Prior to aggregation, we tested consistency for duplicated players: the summed `total_points` from VAA's two matches should equal OLB's `event_points`.

Listing 2: Cross-Dataset Validation

```
# Sum total_points for duplicated pairs
vaa_summed = duplicated_df.groupby(['gw', 'name'])['total_points']
                          .sum().reset_index()

# Merge with OLB event_points
merged = vaa_summed.merge(olb_df[['gw', 'player_name', 'event_points']],
                          left_on=['gw', 'name'],
                          right_on=['gw', 'player_name'])

# Validate: VAA sum == OLB event_points
num_matches = (merged['vaa_sum_total_points'] ==
               merged['event_points']).sum()
```

This check succeeded for all 374 pairs, affirming cross-source reliability and duplicate semantics.

Step 2: Duplicate Aggregation With validity established, aggregation rules preserved key information while eliminating duplicates. Additive metrics (`xP`, `total_points`) were summed across matches. The `starts` field took the maximum to flag any starting appearance. Invariant attributes (`name`, `position`, `team`) retained their first value. For `value` (price) and `minutes`, averages were computed and rounded (to nearest £0.1M and integer). Price averaging accounts for minor intra-week variance without overcomplicating FPL mechanics; minutes averaging prioritizes simplicity over fixture-specific modeling. This condensed the VAA dataset to 27,231 unique rows.

Step 3: Final Duplicate Verification Pre-integration checks confirmed uniqueness:

```
# Verify uniqueness in both datasets
vaa_unique = len(vaa_df.groupby(['gw', 'name'])) == len(vaa_df)
olb_unique = len(olb_df.groupby(['gw', 'player_name'])) == len(olb_df)
# Both returned True
```

Step 4: Dataset Matching Analysis Overlap analysis of (player, gameweek) pairs revealed:

OLB Pairs	VAA Pairs	Matching Pairs	Match %	Only in OLB	Only in VAA
27,658	27,231	27,222	98.43%	436	9

The 98.43% alignment underscores dataset complementarity, despite minor discrepancies from collection timing, late-season additions, or minimal-involvement filtering. An inner join on matches proceeded accordingly.

Step 5: Final Integration Filtered datasets merged on (name, gameweek):

Listing 3: Final Dataset Integration

```
# Filter to matching pairs
matching_pairs = olb_pairs & vaa_pairs

olb_filtered = olb_df[
    olb_df.apply(lambda row: (row['player_name'], row['gw'])
                 in matching_pairs, axis=1)]

vaa_filtered = vaa_df[
    vaa_df.apply(lambda row: (row['name'], row['gw'])
                 in matching_pairs, axis=1)]

# Merge on (name, gw)
integrated_df = pd.merge(olb_filtered, vaa_filtered,
                         left_on=['player_name', 'gw'],
                         right_on=['name', 'gw'],
                         how='inner')
```

The resulting `integrated_df.csv` holds 27,222 records across 16 features. From OLB: `status`, `chance_of_playing_*`, `now_cost`, `event_points`, `ep_*`. From VAA: `position`, `team`, `xP`, `minutes`, `starts`, `total_points`, `value`. Shared keys: `name`, `gw`.

4.2.4 Integration Results and Data Quality

The integration yielded a unified, high-quality dataset spanning the 2024-25 season (GW 1–38), with 27,222 unique player-gameweek records and 98.43% completeness across matching pairs from independent sources. It encompasses 16 features addressing pricing, performance, availability, and predictions, free of duplicates.

Leveraging complementary strengths, the dataset fuses OLB's pricing details with VAA's performance metrics. This synthesis empowers the optimization model to guide both initial squad assembly and ongoing transfer strategies.

The integration phase established a robust foundation for subsequent data cleaning phases, with documented data provenance and validated consistency across sources.

4.3 Phase 2: Data Cleaning

Although data integration merged complementary sources effectively, the resulting `integrated_df.csv` (27,222 records, 16 features) harbored conflicting measurements from independent origins. Cleaning addressed three objectives: reconciling discrepancies, ensuring consistency, and forging unified features for optimization.

Systematic discrepancies clustered in select gameweeks, affirming metric reliability while highlighting vulnerabilities. This phase details reconciliation techniques and validation protocols.

4.3.1 Cost Reconciliation

Pricing data appeared dually: OLB's `now_cost` (scaled ×10) and VAA's `value`, both denoting £0.1M buying prices, potentially captured at disparate intra-gameweek moments.

Of 27,222 records, 1,522 (5.59%) diverged, with differences spanning -2.0 to +2.0 units (±£0.2M; mean 0.022, SD 0.240). Discrepancies peaked in GW2 (70 cases, 11.16%) and GW13 (69 cases, 10.00%).

A conservative strategy adopted the maximum:

Listing 4: Cost Reconciliation Strategy

```
# Create reconciled cost using maximum value
df['cost_reconciled'] = np.maximum(df['now_cost'] * 10, df['value'])
```

This yields cautious estimates, averting selections premised on understated prices, and unifies pricing.

4.3.2 Points Validation

Gameweek points manifested as OLB's `event_points` and VAA's `total_points`, proxying identical FPL earnings.

Exact alignment prevailed across all 27,222 records, validating source fidelity to official scores and integration precision.

4.3.3 Expected Points Reconciliation

Expected points varied: OLB's `ep_this` versus VAA's `xP`.

Of records, 23,515 (86.38%) matched precisely; 3,707 (13.62%) diverged (-22.0 to +26.4; mean -0.061, SD 0.907). Zeros in `xP` afflicted GW22, 32, and 34 entirely, driving peaks: GW32 (520, 65.16%), GW22 (464, 63.65%), GW34 (394, 61.09%). Mismatches occurred in 16 of 38 gameweeks.

Leveraging actual `total_points`, reconciliation favored the lower absolute error:

Listing 5: Expected Points Reconciliation Logic

```
# Calculate absolute differences from actual performance
diff_ep_this = np.abs(df['ep_this'] - df['total_points'])
diff_xP = np.abs(df['xP'] - df['total_points'])

# Select prediction with smaller error
df['ep_reconciled'] = np.where(diff_ep_this <= diff_xP,
                               df['ep_this'],
                               df['xP'])
```

This per-record ensemble selected `ep_this` in 25,487 cases (93.63%) and `xP` in 1,735 (6.37%), varying by gameweek.

Note: This retrospective method suits backtesting but not prospective use, where outcomes are unknown. It mitigates `xP` zeros and outliers, outperforming averages or extrema skewed thereby.

4.3.4 Player Set Standardization

Participation varied: 417 players spanned all 38 gameweeks; 116 in 37; 105 in 36; others, fewer. Retention targeted ≥36 gameweeks, yielding 200 supplementary players for consistency.

Absences for the 221 near-complete players (36-37 gws) clustered non-consecutively in GW1, 15, 29, 34:

Table 1: Distribution of Missing Gameweeks for Players with Near-Complete Participation

GW 1	**GW 15**	**GW 29**	**GW 34**	
7	26	33	50	
GW(1,2)	**GW(1,15)**	**GW(1,34)**	**GW(15,29)**	**GW(29,34)**
11	2	2	28	62

Imputation tailored by variable: Costs interpolated linearly (preceding/following averages, rounded; edges via nearest available) for GW3–38 continuity. Performance metrics (`points`, `eP`, `prob_showup`, `minutes`) zeroed; an `unavailable` flag distinguished imputations.

This culled 166 sporadic players, standardizing to 638 across 38 gameweeks (326 imputations), easing modeling.

4.3.5 Cleaning Results and Data Quality

Cleaning refined `integrated_df.csv` into `cleaned_data.csv`: 24,244 records (ouf of 27222, ‘ 89.0% retention), 638 players/gameweek, fully consistent.

Features: `player_id`, `gw`, `name`, `position`, `team`, `cost` (reconciled), `points`, `eP` (reconciled), `minutes`, `unavailable`.

The final dataset provides a clean, consistent foundation for subsequent modeling phases, with documented reconciliation strategies, validated data quality, and engineered features designed specifically for optimization applications.

4.4 Phase 3: MILP Model Implementation

The Mixed-Integer Linear Programming (MILP) implementation converts the mathematical formulation from Section 3 into executable code via the PuLP library in Python. PuLP was chosen for its declarative constraint syntax, compatibility with diverse solvers (CBC, GLPK, SCIP, CPLEX, Gurobi—both open-source and commercial), and integration with Python’s data tools (pandas, NumPy). It adheres to the two-phase model: Pre-GW1 optimization for initial squad assembly and Post-GW1 for weekly transfers and lineups.

Modularity structures the code into core elements: optimization modules (`pre_gw1_step1.py`, `pre_gw1_step2.py`, `post_gw1_step1.py`) for model execution; a utilities module (`utils.py`) for post-hoc tasks; and a season orchestrator (`run_season_optimizer.py`) overseeing the 38-gameweek simulation. This separation isolates model logic from workflow management, facilitating targeted testing and validation.

4.4.1 Data Structures and Flow

The MILP implementation employs a structured data flow architecture to coordinate optimization decisions while preserving persistent state across gameweeks.

Input Data Structure All optimization functions ingest player data through a pandas DataFrame conforming to the specified schema:

Table 2: Player DataFrame Schema for MILP Optimization

Column	Type	Description
`player_id`	int	Unique identifier (1-638)
`gw`	int	Gameweek number (1-38)
`name`	str	Player name
`position`	str	Position (GK, DEF, MID, FWD)
`team`	str	Club name
`cost`	int	Price in tenths of millions (55 = £5.5m)
`eP`	float	Expected points prediction
`points`	int	Actual points scored (for oracle mode)
`unavailable`	int	Availability flag (0=available, 1=unavailable)

A fundamental implementation invariant requires each DataFrame to encompass precisely one gameweek, devoid of duplicate `player_id` entries. This is enforced via validation at the entry points of all optimization modules: `pre_gw1_step1.py`, `pre_gw1_step2.py`, and `post_gw1_step1.py`.

Listing 6: Single-Gameweek Validation

```
if players_df['player_id'].duplicated().any():
    raise ValueError("More than 1 GW data detected. Each player_id must appear exactly once."
        )
```

State Variables The Post-GW1 optimization sustains four state variables to propagate essential information across gameweeks:

Table 3: State Variables for Post-GW1 Optimization

Variable	Type	Description
`y0`	dict[int → {0,1}]	Current squad (638 binary values)
`p0`	dict[int → int]	Purchase prices in tenths (e.g., 55 = £5.5m)
`B_bank`	int	Cash in bank in tenths (e.g., 50 = £5.0m)
`f`	int	Free transfers available (0-5)

These state variables are initialized following GW1 optimization and updated thereafter after each gameweek in accordance with the transfer decisions.

Decision Variable Output Each optimization yields decision variables as dictionaries mapping player IDs to binary values:

Table 4: Decision Variable Outputs

Variable	Type	Description
x	dict[int → {0,1}]	Starting XI (11 players = 1)
y	dict[int → {0,1}]	Squad (15 players = 1)
c	dict[int → {0,1}]	Captain (1 player = 1)
v	dict[int → {0,1}]	Vice-captain (1 player = 1, post-hoc)
b1-b4	dict[int → {0,1}]	Bench positions (1 per position, post-hoc)

All decision variables encompass the full 638-player set, assigning 0 or 1 to each player ID to enable uniform JSON serialization.

4.4.2 Pre-GW1 Implementation

The Pre-GW1 optimization employs a hierarchical approach.

Step 1: Starter and Captain Selection The initial phase (`pre_gw1_step1.py`) optimizes the starting XI and captaincy under budget and squad constraints. Its objective maximizes expected points from starters, augmented by the captain's double bonus:

Listing 7: Pre-GW1 Step 1 Objective Function

```
# Decision variables
y = pulp.LpVariable.dicts("squad", player_ids, cat='Binary')
x = pulp.LpVariable.dicts("starter", player_ids, cat='Binary')
c = pulp.LpVariable.dicts("captain", player_ids, cat='Binary')
# Objective: Maximize expected points from starters and captain
prob += pulp.lpSum([
    players[pid]['expected_points'] * (x[pid] + c[pid])
    for pid in player_ids
]), "Total_Expected_Points"
```

The constraint set comprehensively enforces all FPL rules governing team selection:

Listing 8: Pre-GW1 Step 1: Comprehensive Constraint Implementation

```
# Lineup constraints: 11 starters with position requirements
prob += pulp.lpSum([x[pid] for pid in player_ids]) == 11, "Lineup_Size"
prob += pulp.lpSum([x[pid] for pid in gk_ids]) == 1, "Starting_GK"
prob += pulp.lpSum([x[pid] for pid in def_ids]) >= 3, "Min_Starting_DEF"
prob += pulp.lpSum([x[pid] for pid in def_ids]) <= 5, "Max_Starting_DEF"
prob += pulp.lpSum([x[pid] for pid in mid_ids]) >= 2, "Min_Starting_MID"
prob += pulp.lpSum([x[pid] for pid in mid_ids]) <= 5, "Max_Starting_MID"
```

```
prob += pulp.lpSum([x[pid] for pid in fwd_ids]) >= 1, "Min_Starting_FWD"
prob += pulp.lpSum([x[pid] for pid in fwd_ids]) <= 3, "Max_Starting_FWD"
# Captain constraints: exactly 1, must be starter
prob += pulp.lpSum([c[pid] for pid in player_ids]) == 1, "One_Captain"
for pid in player_ids:
    prob += c[pid] <= x[pid], f"Captain_Must_Start_{pid}"
# Unavailable players cannot start
for pid in unavailable_ids:
    prob += x[pid] == 0, f"Unavailable_Cannot_Start_{pid}"
# Squad constraints: 15 players with position quotas
prob += pulp.lpSum([y[pid] for pid in player_ids]) == 15, "Squad_Size"
prob += pulp.lpSum([y[pid] for pid in gk_ids]) == 2, "Squad_GK"
prob += pulp.lpSum([y[pid] for pid in def_ids]) == 5, "Squad_DEF"
prob += pulp.lpSum([y[pid] for pid in mid_ids]) == 5, "Squad_MID"
prob += pulp.lpSum([y[pid] for pid in fwd_ids]) == 3, "Squad_FWD"
# Budget constraint: total cost <= 100M (1000 in tenths)
prob += pulp.lpSum([
    players[pid]['cost'] * y[pid] for pid in player_ids
]) <= 1000, "Budget_Limit"
# Club limits: max 3 players per club
for club in clubs:
    club_players = [pid for pid in player_ids if players[pid]['team'] == club]
    prob += pulp.lpSum([y[pid] for pid in club_players]) <= 3, f"Club_Limit_{club}"
# Starters must be in squad
for pid in player_ids:
    prob += x[pid] <= y[pid], f"Starter_In_Squad_{pid}"
```

Upon solving, Pre-GW1 Step 1 yields the optimal starting XI and captain:

Listing 9: Pre-GW1 Step 1 Return Values

```
solution = {
    'x': {pid: int(round(x[pid].varValue)) for pid in player_ids},  # Starters
    'c': {pid: int(round(c[pid].varValue)) for pid in player_ids}   # Captain
    }
```

Step 2: Bench Optimization The subsequent phase (`pre_gw1_step2.py`) optimizes bench composition, fixing the starting XI from Step 1 via equality constraints. This simplifies the constraints to budget, squad quotas, and club limits.

Listing 10: Pre-GW1 Step 2: Reduced Constraint Set

```
# Fix starters from Step 1
for pid in fixed_starter_ids:
    prob += y[pid] == 1, f"Fix_Starter_{pid}"
# Objective: Maximize total expected points of squad
prob += pulp.lpSum([
    players[pid]['expected_points'] * y[pid]
    for pid in player_ids
]), "Total_Squad_Expected_Points"
# Reduced constraints are omitted for simplicity
```

This method optimizes bench composition to enhance squad quality while preserving the optimal starting XI.

Prior to returning the solution, state variables are initialized for GW2:

Listing 11: State Initialization After GW1

```
# Purchase prices: record initial buying prices
p0_buy = {
    pid: (players[pid]['cost'] if y_star[pid] == 1 else 0)
    for pid in player_ids
}
# Bank balance: remaining budget after squad purchase
total_cost = sum(players[pid]['cost'] * y_star[pid] for pid in player_ids)
B_bank = 1000 - total_cost  # 1000 = 100M in tenths
# Free transfers: 2 for GW2 (1 per week + 1 banked)
f = 2
```

The function subsequently returns the optimized squad alongside the initialized state variables:

Listing 12: Pre-GW1 Step 2 Return Values

```
solution = {
    # Final squad
    'y': {pid: int(round(y[pid].varValue)) for pid in player_ids},
    # Initial state for GW2
    'y0': y_star,  # Squad becomes initial state
    'p0': p0_buy,  # Purchase prices
    'B_bank': B_bank,  # Cash in bank
    'f': 2  # Free transfers for GW2
}
```

4.4.3 Post-GW1 Implementation

The Post-GW1 optimization (`post_gw1_step1.py`) implements the transfer-enabled formulation.

Selling Price Calculation FPL's 50% profit lock governs selling prices: upon appreciation, only half the profit is recouped, ceiling-rounded to the nearest £0.1M:

Listing 13: FPL 50% Profit Lock Implementation

```
selling_prices = {}
for pid in player_ids:
    current_price = players[pid]['cost']
    purchase_price = p0[pid]
    if current_price > purchase_price:
        profit = current_price - purchase_price
        # Lock 50% profit, round up (FPL always rounds up)
        selling_prices[pid] = purchase_price + math.ceil(0.5 * profit)
    else:
        # Loss: recover full current price
        selling_prices[pid] = current_price
```

This mechanism ensures that players sold at a gain yield the purchase price plus half the profit, whereas those sold at a loss recover their full current market price.

Dynamic Budget Constraint The budget constraint enforces non-negative cash balances following transfers. Only purchases of new players incur costs, as retained players are already owned:

Listing 14: Post-GW1 Dynamic Budget Constraint

```
# Decision variables for transfers
t = pulp.LpVariable.dicts("transfer_in", player_ids, cat='Binary')
s = pulp.LpVariable.dicts("transfer_out", player_ids, cat='Binary')
# Cost of new purchases <= cash in bank + selling proceeds
prob += pulp.lpSum([
    players[pid]['cost'] * t[pid]
    for pid in player_ids
]) <= B_bank + pulp.lpSum([
    selling_prices[pid] * s[pid]
    for pid in player_ids
]), "Budget_Limit"
```

This constraint dynamically updates the available budget in response to transfer decisions, permitting flexible trading within fiscal limits.

Transfer Logic Constraints Transfer mechanics are enforced via an interconnected suite of constraints:

Listing 15: Transfer Logic Constraints

```
# Squad update: new squad = old squad + buys - sells
for pid in player_ids:
    prob += y[pid] == y0[pid] + t[pid] - s[pid], f"Squad_Update_{pid}"
# No simultaneous buy and sell of same player
for pid in player_ids:
    prob += t[pid] + s[pid] <= 1, f"No_Simultaneous_Transfer_{pid}"
# Can only buy players not in current squad
for pid in player_ids:
    prob += t[pid] <= 1 - y0[pid], f"Buy_Only_Non_Squad_{pid}"
# Can only sell players in current squad
for pid in player_ids:
    prob += s[pid] <= y0[pid], f"Sell_Only_Current_Squad_{pid}"
```

Transfer Penalty Calculation The penalty for excess transfers is calculated at four points per transfer beyond the free allowance:

Listing 16: Extra Transfer Penalty

```
# e: number of extra transfers beyond free_transfers
e = pulp.LpVariable("extra_transfers", lowBound=0, upBound=15, cat='Integer')
# Extra transfers = max(0, total_transfers - free_transfers)
prob += e >= pulp.lpSum([t[pid] for pid in player_ids]) - f, "Extra_Transfers"
# Objective: Expected points minus 4-point penalty per extra transfer
prob += pulp.lpSum([
    players[pid]['expected_points'] * (x[pid] + c[pid])
    for pid in player_ids
]) - 4 * e, "Total_Expected_Points_Minus_Penalty"
```

State Update Following optimization, state variables are updated for the subsequent gameweek:

Listing 17: State Update After Post-GW1 Optimization

```
# Update purchase prices
for pid in player_ids:
    if s_star[pid]:  # Sold player
        p0[pid] = 0
    elif t_star[pid]:  # Bought player
        p0[pid] = players[pid]['cost']
```

```
    # Otherwise unchanged
# Update bank balance
total_selling_proceeds = sum(selling_prices[pid] * s_star[pid]
                             for pid in player_ids)
total_purchase_costs = sum(players[pid]['cost'] * t_star[pid]
                           for pid in player_ids)
B_bank = B_bank + total_selling_proceeds - total_purchase_costs
# Update free transfers (remaining from previous + 1 new per week, max 5)
transfers_used = sum(t_star.values())
remaining_free = max(0, f - transfers_used)
f_new = min(5, remaining_free + 1)
```

4.4.4 Post-Hoc Processing

Vice-captain and bench position assignments follow post-optimization greedy algorithm in `utils.py`.

Vice-Captain Selection The vice-captain is designated as the non-captain starter with the highest expected points:

Listing 18: Vice-Captain Selection

```
vice_captain_id = max(
    (pid for pid in player_ids if x_star[pid] == 1 and c_star[pid] == 0),
    key=lambda pid: players[pid]['expected_points']
)
```

Bench Position Assignment Bench ordering adheres to FPL substitution priorities:

Listing 19: Bench Position Assignment

```
# Position 1: Backup goalkeeper
bench_gks = [pid for pid in bench_ids if players[pid]['position'] == 'GK']
gk_id = bench_gks[0]
# Positions 2-4: Outfield sorted by expected points (descending)
outfield_ids = [pid for pid in bench_ids if players[pid]['position'] != 'GK']
outfield_sorted = sorted(outfield_ids,
                         key=lambda pid: players[pid]['expected_points'],
                         reverse=True)
# Bench order: [pos1_GK, pos2_best, pos3_mid, pos4_worst]
bench_order = [gk_id] + outfield_sorted
```

4.4.5 Season Orchestration

The season optimizer (`run_season_optimizer.py`) orchestrates optimization across all 38 gameweeks.

Oracle Mode In oracle mode, conditional expected points are substituted with actual points scored for each player. This facilitates perfect-information backtesting, yielding performance upper bounds.

Listing 20: Oracle Mode Implementation

```
players_df = players_df.copy()
if oracle:
    # Oracle mode: replace expected points with actual points
    players_df['expected_points'] = players_df['points']
else:
    # Normal mode: use predicted expected points
    players_df['expected_points'] = players_df['eP']
```

Workflow Management The orchestrator oversees the complete workflow:

Listing 21: Season Optimization Workflow

```
for gw in range(1, 39):
    players_gw = all_data[all_data['gw'] == gw].copy()
    if gw == 1:
        # Pre-GW1: Three-step pipeline
        step1 = optimize_pre_gw1_step1(players_gw, solver=solver)
        step2 = optimize_pre_gw1_step2(players_gw, step1['x'], solver=solver)
        post_hoc = process_pre_gw1_solution(players_gw, step1['x'],
                                            step2['y'], step1['c'])
        # Initialize state variables
        y0, p0, B_bank, f = step2['y0'], step2['p0'], step2['B_bank'], step2['f']
    else:
        # Post-GW1: Two-step pipeline
        step1 = optimize_post_gw1_step1(players_gw, y0, p0, f, B_bank,
                                        solver=solver)
        post_hoc = process_post_gw1_solution(players_gw, step1['x'],
                                             step1['y'], step1['c'])
        # Update state variables
        y0, p0, B_bank, f = step1['y0'], step1['p0'], step1['B_bank'], step1['f']
```

4.4.6 Technical Challenges and Solutions

Floating-Point Precision PuLP solvers return binary variables as floating-point values (e.g., 0.9999999 instead of 1.0). This caused errors in downstream processing expecting exact integers. Rounding to the nearest integer resolves this:

Listing 22: Binary Variable Precision Handling

```
# Extract optimal solution and force to exact integers
x_star = {pid: int(round(x[pid].varValue)) for pid in player_ids}
y_star = {pid: int(round(y[pid].varValue)) for pid in player_ids}
```

Solver Selection and Compatibility A unified interface supports multiple solvers:

Listing 23: Solver Selection Interface

```
def get_pulp_solver(solver_name='CBC', msg=0):
    solvers = {
        'CBC': pulp.PULP_CBC_CMD(msg=msg),
        'GLPK': pulp.GLPK_CMD(msg=msg),
        'SCIP': pulp.SCIP_PY(msg=msg)
    }
    return solvers.get(solver_name.upper(), pulp.PULP_CBC_CMD(msg=msg))
```

4.4.7 Implementation Results

The MILP implementation sequentially optimizes all 38 gameweeks, with each phase leveraging state variables from the prior one. Its modular architecture facilitated isolated validation of components, achieving full test coverage for Pre-GW1 and Post-GW1 pipelines.

Optimization proceeded across three solvers (CBC, GLPK, SCIP) to verify independence and robustness. All generated consistent results, affirming the mathematical model's accuracy.

The output comprises a standardized JSON file, wherein each gameweek encompasses 12 keys: eight binary decision dictionaries (`x`, `y`, `c`, `v`, `b1`, `b2`, `b3`, `b4`)—each mapping all 638 player IDs to {0,1}—plus four state variables (`y0` dict, `p0` dict, `B_bank` scalar, `f` scalar). This uniform structure promotes data consistency across gameweeks and facilitates integration with backtesting and analytical tools.

This implementation underpins backtesting evaluations and comparisons with CP formulations.

4.5 Phase 4: CP MiniZinc Implementation

The Constraint Programming (CP) formulation, realized in MiniZinc, presents an alternative to MILP for probing varied paradigms and solvers. MiniZinc, a declarative language tailored for constraint satisfaction and optimization, supports this modeling. The implementation accommodates three CP solvers—OR-Tools CP-SAT (default), Gecode, and Chuffed.

The primary rationale for a parallel CP formulation alongside MILP is to evaluate alternative modeling paradigms and corroborate solution integrity across independent solver frameworks. MiniZinc's declarative syntax enables set-based representations and global constraints, while CP solvers emphasize constraint propagation over MILP's linear relaxations. The CP implementation mirrors the MILP's modular architecture, comprising three optimization modules for Pre-GW1 Step 1, Pre-GW1 Step 2, and Post-GW1 phases. Each module integrates a Python wrapper (`.py`) for data conversion and solver interfacing with a MiniZinc model (`.mzn`) for declarative constraint formulation. This bifurcation permits MiniZinc's domain-specific syntax to encode optimization logic, while Python oversees data preparation and result extraction.

4.5.1 Pre-GW1: Set-Based Formulation

The initial step optimizes the starting XI and captain selection, paralleling the MILP Pre-GW1 Step 1 but employing set-based decision variables in place of binary arrays.

Set Variables In the Pre-GW1 Step 1 model, the core decision variables are defined as sets rather than indexed binary variables:

Listing 24: MiniZinc Set-Based Decision Variables (pre_gw1_step1.mzn)

```
% Decision Variables: Sets and Set-of-Sets (Partitions)
var set of Players: Squad;          % Q: Squad set, |Q|=15
var set of Players: Starters;       % S subset Q: Starting XI, |S|=11
var set of Players: Captain;        % Singleton set for captain

array[Position] of var set of Players: SquadPartitions;
array[Position] of var set of Players: StarterPartitions;

% Subset Constraints
constraint Starters subset Squad;
constraint Captain subset Starters;
```

This formulation encapsulates the problem structure directly: the squad comprises a set of players, the starters a subset of the squad, and the captain a singleton subset of the starters. Set membership is articulated through subset relationships.

Partitioning into Disjoint Sets by Position To enforce FPL positional requirements, the model partitions the squad and starting XI into disjoint position-specific subsets. A partition consists of mutually exclusive subsets that collectively encompass the entire set, assigning each player to exactly one position category.

This structure is maintained through two constraint types: disjointness, ensuring subset exclusivity, and cardinality, specifying subset sizes. Disjointness constraints guarantee that no player appears in multiple position groups:

Listing 25: Disjoint Partition Constraints (pre_gw1_step1.mzn)

```
% Partition Constraints (disjoint union, subsets of available players)
```

```
constraint forall(p1, p2 in Position where p1 < p2) (disjoint(SquadPartitions[p1],
    SquadPartitions[p2]));
constraint Squad = array_union([SquadPartitions[p] | p in Position]);
constraint forall(p in Position) (SquadPartitions[p] subset PosPlayers[p]);

constraint forall(p1, p2 in Position where p1 < p2) (disjoint(StarterPartitions[p1],
    StarterPartitions[p2]));
constraint Starters = array_union([StarterPartitions[p] | p in Position]);
constraint forall(p in Position) (StarterPartitions[p] subset SquadPartitions[p]);
```

The `disjoint()` constraint guarantees that no player appears in multiple position sets, while `array_union()` reconstructs the complete squad and starting XI from their position-based partitions.

Cardinality FPL squad and starting XI composition requirements across four positions are enforced through cardinality constraints on each partition. These are implemented via the `card()` function, which computes the number of elements in a set:

Listing 26: Cardinality Constraints for FPL Rules (pre_gw1_step1.mzn)

```
% Cardinalities
constraint card(Squad) = 15;
constraint card(Starters) = 11;
constraint card(Captain) = 1;
% Position-Specific Cardinalities
constraint card(SquadPartitions[GK]) = 2 /\ card(StarterPartitions[GK]) = 1;
constraint card(SquadPartitions[DEF]) = 5 /\ 3 <= card(StarterPartitions[DEF]) /\ card(
    StarterPartitions[DEF]) <= 5;
constraint card(SquadPartitions[MID]) = 5 /\ 2 <= card(StarterPartitions[MID]) /\ card(
    StarterPartitions[MID]) <= 5;
constraint card(SquadPartitions[FWD]) = 3 /\ 1 <= card(StarterPartitions[FWD]) /\ card(
    StarterPartitions[FWD]) <= 3;
```

Integer Arithmetic and Scaling The MiniZinc implementation mandates integer arithmetic exclusively. Expected points are scaled by a factor of 10 to eliminate fractions.

The Python data preparation module (`data_prep.py`) manages these scalings; post-solving, objective values are rescaled to original units for analysis.

Step 2: Bench Optimization The subsequent step optimizes bench composition, with the starting XI from Step 1 held fixed, paralleling the MILP Pre-GW1 Step 2 formulation.

Listing 27: Pre-GW1 Step 2: Fixed Starters Constraint (pre_gw1_step2.mzn)

```
% Fix Starters and Partitions from Step 1
constraint Starters = FixedStarters;
constraint forall(p in Position) (StarterPartitions[p] = FixedStarterPartitions[p]);
```

This optimization designates the Step 1 starters as mandatory squad members, then selects 4 additional players under the identical constraints, with the 11 starters pre-fixed.

4.5.2 Post-GW1: Transfer-Enabled Optimization

The Post-GW1 model incorporates added complexity via transfer mechanics. An initial set-difference method (deriving buys as `Squad diff OldSquad`) was explored but abandoned due to unresolved implementation challenges; the final formulation employs explicit binary variables for transfers to guarantee reliable constraint propagation:

Listing 28: Explicit Transfer Variables (post_gw1_step1.mzn)

```
% Explicit Transfer Variables
array[Players] of var bool: t;  % t[i] = 1 if buy i (transfer in)
array[Players] of var bool: s;  % s[i] = 1 if sell i (transfer out)
% Squad update: y_i = y0_i + t_i - s_i
constraint forall(i in Players) (
    y[i] = bool2int(y0[i]) + bool2int(t[i]) - bool2int(s[i])
);
```

This formulation maintains compatibility with the MILP model's transfer semantics while leveraging MiniZinc's boolean type system.

Set-Array Channeling A key technical challenge in the Post-GW1 model involves aligning set-based squad representations with array-based transfer variables. MiniZinc's channeling constraints establish bidirectional consistency between these paradigms:

Listing 29: Set-Array Channeling (post_gw1_step1.mzn)

```
% Decision variables in both representations
array[Players] of var 0..1: y;  % Squad membership as array
var set of Players: Squad;       % Squad as set
% Channeling: link int array y with set variable Squad
constraint forall(i in Players) (
    (y[i] = 1) <-> (i in Squad)
);
```

The equivalence constraint `(y[i] = 1) <-> (i in Squad)` links binary array element $y[i]$ to squad membership: player i resides in the Squad set if and only if $y[i] = 1$. This channeling permits simultaneous application of set-based constraints—for subset relations and global rules—with array-based constraints for transfer computations.

Transfer Constraints Transfer logic constraints parallel the MILP formulation, adapted to MiniZinc's boolean type system:

Listing 30: Transfer Logic (post_gw1_step1.mzn)

```
% Cannot buy and sell the same player
constraint forall(i in Players) (
    not (t[i] /\ s[i])
);
% Can only buy players not in current squad
constraint forall(i in Players) (
    y0[i] -> not t[i]
);
% Can only sell players in current squad
constraint forall(i in Players) (
    not y0[i] -> not s[i]
);
```

The logical implication operator `->` enables more concise constraint formulations than the inequality-based encodings required in MILP.

4.5.3 Python-MiniZinc Integration

The implementation leverages the `minizinc` Python package to integrate Python's data processing ecosystem with MiniZinc's constraint solver.

Model Loading and Solver Selection Each optimization module loads the corresponding MiniZinc model and instantiates a solver:

Listing 31: MiniZinc Model and Solver Initialization

```
from minizinc import Instance, Model, Solver
# Load the MiniZinc model
model_path = os.path.join(os.path.dirname(__file__), 'pre_gw1_step1.mzn')
model = Model(model_path)
# Get solver (supports cp-sat, gecode, chuffed)
try:
    solver = Solver.lookup('cp-sat')  # OR-Tools CP-SAT (default)
```

```
except LookupError:
    solver = Solver.lookup('gecode')  # Fallback to Gecode
# Create instance
instance = Instance(solver, model)
```

OR-Tools CP-SAT was used as the default solver, with Gecode and Chuffed also tested for validation.

4.5.4 Technical Challenges and Solutions

Parameter Conversion and Array Indexing A key technical nuance is MiniZinc's 1-based array indexing, contrasting Python's 0-based convention. The data preparation module manages this conversion:

Listing 32: 1-Based Array Construction (data_prep.py)

```
# Build arrays with sequential 1-based indices
expected_points = [0] * (n_players + 1)  # Index 0 unused
cost = [0] * (n_players + 1)
unavailable = [False] * (n_players + 1)
for pid in player_ids:
    idx = player_id_to_idx[pid]  # 1-based index
    row = players_df[players_df['player_id'] == pid].iloc[0]
    expected_points[idx] = int(round(float(row['expected_points']) * 10))
    cost[idx] = int(round(float(row['cost'])))
    unavailable[idx] = bool(row['unavailable'])
# Return arrays excluding index 0 (MiniZinc expects 1..n_players)
params = {
    'expected_points': expected_points[1:],  # 1..n_players
    'cost': cost[1:],
    'unavailable': unavailable[1:]
}
```

Result Parsing and Set Conversion Post-solving, MiniZinc yields set variables as Python sets of 1-based indices, which require conversion to binary dictionaries mapping player IDs to 0/1 values:

Listing 33: Set to Binary Dictionary Conversion (data_prep.py)

```
# Get sets from MiniZinc output
squad_set = result['Squad']          # Set of 1-based indices
starters_set = result['Starters']
captain_set = result['Captain']
# Create mapping from 1-indexed position to player_id
player_ids = players_df['player_id'].tolist()
```

```
# Convert sets to binary dictionaries
y = {pid: (1 if (idx + 1) in squad_set else 0)
     for idx, pid in enumerate(player_ids)}
x = {pid: (1 if (idx + 1) in starters_set else 0)
     for idx, pid in enumerate(player_ids)}
c = {pid: (1 if (idx + 1) in captain_set else 0)
     for idx, pid in enumerate(player_ids)}
```

This conversion aligns the CP output with the MILP format, facilitating uniform post-hoc processing and backtesting pipelines.

Upper Bound Calculation CP solvers leverage tight upper bounds on objective variables to accelerate search processes. The data preparation module derives problem-specific bounds accordingly.

Listing 34: Upper Bound Calculation (data_prep.py)

```
def calculate_upper_bound(players_df):
    # Get available players only
    available_df = players_df[players_df['unavailable'] == 0].copy()

    # Best 11 starters by expected points
    best_11 = available_df.nlargest(11, 'expected_points')['expected_points'].sum()
    captain_bonus = available_df['expected_points'].max()

    ub = best_11 + captain_bonus
    # Scale by 10 for integer arithmetic, add buffer
    return int(ub * 10) + 10
```

This optimistic estimate assumes selecting the 11 highest-expected-points players, disregarding positional and squad constraints. Though unattainable, it furnishes a reliable upper limit for effective pruning.

4.5.5 Implementation Results

The MiniZinc CP implementation satisfied all validation criteria: comprehensive optimization across 38 gameweeks with persistent state, and output aligned with the MILP format to facilitate direct comparison. Execution encompassed three solvers (OR-Tools CP-SAT—default—Gecode, Chuffed), affirming independence across backends.

4.6 Phase 5: Backtesting and Validation

The backtesting phase evaluates optimizer performance against actual gameweek outcomes, incorporating basic FPL rule validation as a sanity check. Its core objective is to quantify practical efficacy of optimization strategies, rather than aiding the optimization itself. Users may backtest custom solutions, provided they adhere to standardization (638-player dataset and JSON format).

The backtester ingests season-long outputs (JSON files with 38 gameweek decisions) and emulates gameplay via FPL's substitution rules, captain doubling, and transfer tracking. This yields metrics such as total points, per-gameweek scores, and transfer efficiency, alongside compliance verification.

Implementation comprises three modules: `fpl_validator.py` for rule checks, `fpl_point_calculator.py` for substitutions and scoring, and `backtester.py` for orchestrating the full season. This modularity decouples validation from computation, supporting isolated testing and cross-optimizer reuse.

4.6.1 Architecture and Data Flow

The backtester operates on two core data sources: optimizer output JSON files and the cleaned dataset.

Input: Optimizer Output Format Each optimizer generates a standardized JSON file encapsulating decisions for all 38 gameweeks. Per gameweek, it features 12 keys:

Table 5: Optimizer Output Structure Per Gameweek

Key	Type	Description
`x`	dict[player_id → {0,1}]	Starting XI (11 players = 1, rest = 0)
`y`	dict[player_id → {0,1}]	Squad (15 players = 1, rest = 0)
`c`	dict[player_id → {0,1}]	Captain (1 player = 1, rest = 0)
`v`	dict[player_id → {0,1}]	Vice-captain (1 player = 1, rest = 0)
`b1-b4`	dict[player_id → {0,1}]	Bench positions (1 player per position)
`y0`	dict[player_id → {0,1}]	Previous squad state
`p0`	dict[player_id → int]	Purchase prices (tenths of millions)
`B_bank`	int	Cash in bank (tenths of millions)
`f`	int	Free transfers available (0-5)

All binary decision variables (`x`, `y`, `c`, `v`, `b1`–`b4`, `y0`) manifest as dictionaries mapping the 638 player IDs (strings) to {0, 1}, ensuring structural uniformity across gameweeks.

Input: Player Performance Data The cleaned dataset (`cleaned_data.csv`) provides actual gameweek performance metrics for all players. For each gameweek, the backtester extracts relevant records and merges them with optimizer decisions via `player_id`.

Output: Detailed Gameweek Reports The backtester produces detailed text reports featuring per-gameweek analyses of points scored, transfer activities, captain selections, automatic substitutions, active lineup configurations, and bench statuses. Concluding summaries aggregate season-long metrics, including total points over 38 gameweeks, cumulative transfers, and average points per gameweek.

4.6.2 FPL Rule Validation

The `fpl_validator.py` module employs a comprehensive single-pass algorithm to validate all FPL team selection rules.

Validation Algorithm The validation function ingests gameweek data and decision variables, executing checks in logical sequence. It commences with squad size verification (precisely 15 players), followed by positional composition (2 GK, 5 DEF, 5 MID, 3 FWD). Club affiliation limits are then assessed, counting players per team to enforce a maximum of 3 per club. For the starting XI, it confirms size (exactly 11 players), formation validity (1 GK, 3–5 DEF, 2–5 MID, 1–3 FWD), and bench setup (position 1: GK; positions 2–4: outfield). Concluding checks validate captaincy (captain $\neq$ vice-captain; both starters).

Formation Validation During Substitutions A dedicated validation function upholds legal formations following automatic substitutions. It assesses potentially incomplete lineups (fewer than 11 players) through positional counts: precisely 1 GK, 3–5 DEF, 2–5 MID, and 1–3 FWD. Integral to the substitution algorithm, it confirms validity after each prospective bench addition, preventing illegal configurations.

4.6.3 Points Calculation

The `fpl_point_calculator.py` module implements FPL's automatic substitution mechanism and captain doubling rules.

Captain Bonus Application Following substitutions, the final active lineup determines the base points as the sum of all active players' scores. Captain doubling then applies, with fallback to the vice-captain: if the captain played (non-zero minutes and active), their points are added once more to the total. Otherwise, if the vice-captain played, their points receive the doubling instead. This ensures precisely one bonus, favoring the captain but defaulting to the vice-captain as needed.

4.6.4 Season Orchestration

The `backtester.py` module orchestrates season-long processing across all 38 gameweeks.

Gameweek Processing Loop For each gameweek, the backtester follows a standardized workflow: extracting team selections from binary dictionaries (squad, starters, captain, vice-captain, bench positions), retrieving actual performance data from the cleaned dataset, validating selections against FPL rules, and computing points via automatic substitutions. Cumulative points are tracked season-long.

Transfer Detection Transfers are identified by comparing consecutive gameweek squads: bought players appear in the current squad but not the previous, while sold players appear in the previous but not the current. Set difference operations facilitate this detection efficiently.

Listing 35: Transfer Detection Algorithm

```python
def detect_transfers(current_squad: List[int],
                     previous_squad: List[int]) -> Tuple[List, List]:
    """Detect transfers using set difference."""
    current_set = set(current_squad)
    previous_set = set(previous_squad)
    transfers_in = list(current_set - previous_set)
    transfers_out = list(previous_set - current_set)
    return transfers_in, transfers_out
```

Output Generation For each gameweek, the backtester produces detailed formatted output commencing with a header denoting the gameweek number, followed by points scored that week and cumulative totals. The transfer section enumerates incoming players (if applicable), detailing their name, ID, position, and team. Subsequent sections cover captain and substitution particulars, an active lineup breakdown with per-player points, and bench status indicating utilized or idle reserves. Upon completing all 38 gameweeks, a final summary delivers season aggregates: total points, transfer count, average points per gameweek, and verification of full-season execution.

4.6.5 Technical Challenges and Solutions

The backtester joins optimizer decisions (via player IDs) with actual performance data from `cleaned_data.csv`, where ID mismatches would precipitate lookup failures. It relies on the uniform `player_id` scheme from data cleaning, assigning identical IDs to all 638 players across outputs and the dataset. This permits direct DataFrame filtering for gameweek data, followed by squad-specific extraction from decisions. For external users, adherence to this 638-player mapping is requisite; alternatively, forking the repository and implementing extensions is advised to accommodate custom datasets.

4.6.6 Implementation Results

The backtesting phase fulfilled its dual aims of performance evaluation and correctness verification. Detailed per-gameweek reports illuminate optimizer decisions, highlighting patterns in transfer activity, captaincy, and substitutions. These observations drove refinements to the optimization model and affirmed the accuracy of implemented FPL rules against official mechanics. The backtester establishes a basis for comparative evaluations of optimization paradigms and strategies.

5 Evaluation

5.1 Evaluation Methodology

The optimization system underwent comprehensive backtesting across the full 2024-25 Fantasy Premier League season (38 gameweeks). Evaluation encompassed two prediction scenarios:

- **Oracle scenario**: Employs actual gameweek outcomes as predictions, simulating perfect foresight. This validates algorithmic correctness and delineates an upper performance bound, yielding particularly insightful analytical benchmarks.
- **Normal scenario**: Leverages expected points from integrated sources (calibrated per the implementation section). This assesses the pipeline's efficacy with external predictions.

Performance was quantified via four metrics: seasonal total points, GW1 points (gauging initial squad efficacy), total transfers executed, and average solve time per instance.

5.2 Experimental Setup

Evaluation encompassed six solver configurations: three MILP solvers (CBC, GLPK, SCIP) via PuLP, and three CP solvers (CP-SAT, Chuffed, Gecode) via MiniZinc. MILP instances resolved expeditiously. CP solvers were assessed under two timeout regimes—10 seconds and 60 seconds—to quantify computational tradeoffs.

The oracle scenario spanned all six solvers, facilitating algorithmic comparisons under perfect foresight. The normal scenario was confined to MILP solvers, as CP variants yielded impractical runtimes despite extended time limits. Benchmarking against real FPL manager rankings drew from the FPL community website, a reliable source given the prominence of top performers within the community.

5.3 Results

Table 6 presents the comprehensive performance comparison across all solver configurations and scenarios.

Approach	Solver	Scenario	GW1 Points	Total Points	Transfers	Avg Time (sec)
MILP	CBC	Oracle	131	5548	280	0.20
MILP	CBC	Normal	71	3152	91	0.14
MILP	GLPK	Oracle	131	5579	287	0.12
MILP	GLPK	Normal	62	3206	88	0.14
MILP	SCIP	Oracle	131	5597	291	0.20
MILP	SCIP	Normal	76	3232	92	0.24
CP	CP-SAT	Oracle (10s)	99	3417	145	$\sim$10.0*
CP	CP-SAT	Oracle (60s)	131	3911	178	$\sim$60.0**
CP	Chuffed	Oracle (10s)	56	2291	75	$\sim$10.0*
CP	Chuffed	Oracle (60s)	74	2531	102	$\sim$60.0*
CP	Gecode	Oracle (10s)	77	2732	74	$\sim$10.0*
CP	Gecode	Oracle (60s)	83	2935	91	$\sim$60.0*

*Solver reached timeout limit on most gameweeks

**CP-SAT achieved optimal solution for GW1 in 57 seconds, then hit timeout on later gameweeks

Table 6: Performance comparison across all solver configurations. Oracle scenario uses perfect foresight of actual gameweek outcomes; Normal scenario uses expected points from integrated data sources.

5.4 Results Interpretation

To contextualize these results, actual FPL manager rankings from the 2024-25 season (sourced from the official FPL community website) indicate that the top-ranked manager amassed 2810 points, with rank 10 at 2754, rank 1000 at 2657, and rank 10,000 at 2599. In the normal scenario, MILP solvers yielded substantially superior totals (CBC: 3152; GLPK: 3206; SCIP: 3232 points), surpassing the elite human benchmark. This discrepancy implies overfitting in the expected points data—likely incorporating post-deadline information unavailable in real-time decision-making. Oracle outcomes (CBC: 5548; GLPK: 5579; SCIP: 5597 points) approximate double the top human performance.

The oracle scenario affirms algorithmic fidelity: with flawless predictions, all three MILP solvers consistently deliver optimal or near-optimal solutions, varying by at most 1% from one another. Notably, multiple optima per gameweek—especially in bench selection, which indirectly influences budgets and transfers—engender subtle divergences. These propagate as butterfly effects through subsequent gameweeks, accounting for inter-solver point variances despite individual optimality.

5.5 Comparative Analysis

5.5.1 MILP vs CP Performance

In the oracle scenario, the leading MILP solver (SCIP: 5597 points) surpassed the top CP solver (CP-SAT with 60-second timeout: 3911 points) by 1686 points. MILP solvers attained these outcomes in under 0.25 seconds on average, whereas CP solvers exhausted the 60-second timeouts without consistently reaching optimality.

CP-SAT resolved GW1 optimally in 57 seconds, illustrating that CP solvers could theoretically achieve full optimality with unconstrained time. Nonetheless, at 240 times the average MILP runtime (0.25 seconds per gameweek), CP's performance renders it impractical for real-time applications. This disparity justifies forgoing the initially proposed hybrid MILP/CP decomposition.

5.5.2 Transfer Behavior

Transfer patterns illustrate the influence of prediction quality on optimization strategy. In the oracle scenario, MILP solvers executed 280–291 transfers (averaging 7.4–7.7 per week), compared to 88–92 transfers (averaging 2.3–2.4 per week) in the normal scenario. This disparity stems from prediction scale and confidence, rather than accuracy alone.

Under perfect foresight (oracle), expected point gains from weekly squad adjustments routinely surpass the 4-point transfer penalty, fostering aggressive transfers. Conversely, the conservative expected points in the normal scenario yield marginal or negative net gains for excess transfers, promoting preservation of free allowances. Thus, the optimization adaptively responds to input prediction magnitudes: substantial differentials justify additional transfers, while subdued ones favor restraint.

For CP solvers, transfer counts (74-178) cannot be meaningfully interpreted as the search was incomplete within the timeout limits.

5.6 Success Analysis

The evaluation demonstrates three key successes:

Practical Modeling The oracle scenario's attainment of roughly double the points of the top human manager demonstrates that the optimization problem does not constitute a bottleneck in the FPL game, enabling substantial performance gains under perfect foresight.

Computational Efficiency MILP solvers resolved instances in under 0.25 seconds consistently, supporting real-time deployment in interactive settings. This performance underscores the tractability of the hierarchical decomposition and transfer dynamics modeling.

5.7 Limitations and Shortcomings

Constraint Programming Timeout Issues CP solvers encountered severe timeouts, routinely exhausting the 10–60 second limits without completing searches. Although CP-SAT resolved GW1 optimally in 57 seconds—indicating theoretical feasibility, this latency is untenable for real-time applications demanding sub-second responses. At 240 times MILP's average speed, CP formulations prove unsuitable for production environments.

Multiple Optima Effects Multiple optimal solutions exist in each gameweek's optimization, particularly for bench selection. Although bench players do not influence the objective function, they impact budget allocation and subsequent transfer options. Consequently, minor differences in bench composition propagate as cascading effects through later gameweeks. This accounts for the subtle variations in season-long point totals across solvers, each of which identifies optimal solutions on a per-gameweek basis. Absent simultaneous multi-week modeling, such trajectory divergences remain inherent.

6 Conclusion and Critical Appraisal

This project has developed and evaluated an optimization framework for Fantasy Premier League squad management, leveraging both Mixed Integer Linear Programming (MILP) and Constraint Programming (CP) paradigms. The following discussion critically examines the project's accomplishments, limitations, and prospective directions.

6.1 Summary of Accomplishments

The project successfully delivered an end-to-end FPL optimization system spanning the complete 2024-25 season (38 gameweeks). The key accomplishments include:

Data Engineering: Integrated and cleaned 27,222 player-gameweek records from three independent sources, reconciling schema inconsistencies, resolving 374 duplicate fixtures, and applying strategies for conflicting measurements.

Mathematical Modeling: Developed comprehensive MILP and CP formulations that precisely model FPL's intricate rules, including squad composition constraints, budget dynamics with 50% profit lock, hierarchical starter-bench optimization, transfer mechanics with free banking, and automatic substitutions.

Multi-Paradigm Implementation: Achieved successful deployment via PuLP for MILP (with CBC, GLPK, and SCIP backends) and MiniZinc for CP (with CP-SAT, Gecode, and Chuffed backends), facilitating rigorous comparative evaluation.

Validation and Evaluation: Developed complete backtesting infrastructure with dual scenarios—oracle mode yielding approximately double human benchmark performance (5,548–5,597 points versus 2,810 for top manager)—and exhaustive FPL rule validation across all 38 gameweeks.

6.2 Strengths of the Approach

The project demonstrated several notable strengths that contribute to its practical value and methodological rigor.

Computational Efficiency: MILP solvers demonstrated superior computational efficiency, resolving each gameweek optimization in 0.12–0.24 seconds on average. This sub-second latency supports real-time deployment in interactive applications.

Cross-Platform Accessibility: Technology selections emphasize ease of use and cross-platform accessibility. PuLP furnishes a pure-Python MILP environment and accommodates open-source solvers including CBC, GLPK, and SCIP. MiniZinc supplies a well-documented constraint modeling language with Python integration. Both tools deploy effortlessly across Windows, macOS, and Linux, avoiding complex compilations or licensing barriers that might constrain adoption among researchers and practitioners.

Modular Architecture: The implementation's modular architecture supports independent validation and testing. Distinct optimization modules for Pre-GW1 Step 1, Pre-GW1 Step 2, and Post-GW1 enable targeted unit testing of each phase. The utility module (`utils.py`) encapsulates post-hoc processing, segregating vice-captain and bench ordering logic from the core optimization. The season orchestrator (`run_season_optimizer.py`) oversees state management without entanglement to solver specifics. This separation of concerns expedited development by permitting incremental validation and debugging.

6.3 Weaknesses and Limitations

Despite its strengths, the project exhibits significant limitations that constrain both theoretical performance and practical applicability.

Scope Limitations: The one-week forward horizon fundamentally constrains optimization efficacy. By addressing each gameweek in isolation—relying exclusively on immediate next-week predictions—the model overlooks multi-period dynamics, such as anticipating future price escalations (e.g., acquiring rising players preemptively), favorable fixture sequences (e.g., conserving transfers for upcoming runs), and chip deployment (e.g., timing the wildcard for unlimited free transfers).

Constraint Programming Performance Gap: CP solvers demonstrated severe performance limitations, rendering them impractical for this domain. Although CP-SAT resolved GW1 optimally in 57 seconds, indicating theoretical parity with unbounded time, such latencies preclude real-time deployment requiring sub-second responses.

Data Quality Challenges: The data integration phase uncovered substantial quality challenges in the open datasets,

including discrepancies in player prices and expected points across sources, as well as complete absences of predictions for certain gameweeks. These inconsistencies, compounded by the lack of centralized quality control in community-maintained repositories, highlighted the inherent difficulties of merging independent data streams. Additionally, variability in player participation across gameweeks necessitated imputation for standardization, which introduced potential distortions in representing actual availability, pricing, and performance dynamics.

Multiple Optima and Trajectory Divergence: Multiple optimal solutions for individual gameweeks engendered unanticipated variations in season-long performance. As bench players do not directly influence the objective function, diverse bench compositions yield equivalent gameweek scores; yet, these choices subtly alter budget allocation and future transfer options, precipitating butterfly effects that propagate through subsequent gameweeks.

6.4 Future Work and Extensions

Several directions could extend and improve the current work:

Multi-Period Rolling Horizon Optimization: Extend the one-week horizon to 2–4 weeks via rolling optimization: solve multi-period problems but implement only the first week's decisions, capturing price rises, fixture runs, and chip timing.

Risk-Aware or Non-Linear Objective Functions: Incorporate risk via minimal-variance optimization constraints to model manager preferences on conservative low-variance squads using MINLP/MIQP solvers.

Chip Strategy Optimization: Model FPL chips—Wildcard (unlimited transfers), Bench Boost (bench scoring), Triple Captain (tripled points), Free Hit (temporary squad)—using multi-period binary variables constrained by usage limits.

References

[1] Adrian Becker and Xu Andy Sun. An analytical approach for fantasy football draft and lineup management. *Journal of Quantitative Analysis in Sports*, 12(1):17–30, 2016.

[2] Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Gregory Gange, and Kent Francis. Chuffed: A lazy clause generation solver. `https://github.com/chuffed/chuffed`, 2018.

[3] John Forrest and Lou Hafer. Cbc user guide. `https://www.coin-or.org/Cbc/cbcuserguide.html`, 2023. COIN-OR Foundation.

[4] Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leona Eifler, Mathieu Gasse, et al. The scip optimization suite 10.0. *arXiv preprint arXiv:2511.18580*, 2025.

[5] Daniel Groos. Openfpl: An open-source forecasting method rivaling state-of-the-art fantasy premier league services, 2025.

[6] Andrew O. Makhorin. Glpk (gnu linear programming kit), version 5.0. `http://www.gnu.org/software/glpk/`, 2024. GNU Project.

[7] Stuart Mitchell, Michael O'Sullivan, and Iain Dunning. Pulp: A linear programming toolkit for python. In *Operations Research in Python (ORPY)*, 2011.

[8] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543. Springer, 2007.

[9] olbauday. Fpl-elo-insights. `https://github.com/olbauday/FPL-Elo-Insights`, 2025. FPL data fused with Elo ratings for teams, including gameweek-split CSV files (GW1-GW38) for the 2024-2025 season.

[10] Laurent Perron and Vincent Furnon. Or-tools version 9.9: Google's operations research tools. `https://developers.google.com/optimization`, 2025. Google.

[11] Danial Ramezani and Tai Dinh. A data-driven framework for team selection in fantasy premier league, 2025.

[12] Randdalf. fplcache. `https://github.com/Randdalf/fplcache`, 2024. Daily cache of Fantasy Premier League bootstrap data from the official API in LZMA-compressed JSON format.

[13] Christian Schulte and Mikael Z Lagerkvist. Gecode 6.2 user manual. `https://www.gecode.org/`, 2023.

[14] vaastav. Fantasy-premier-league. `https://github.com/vaastav/Fantasy-Premier-League`, 2025. CSV datasets for Fantasy Premier League 2024-25 season, including player statistics, gameweek-specific data, team information, and season histories.