Rethinking MILP as a Policy
Embedding MILP in Sequential Decision Analytics

Adam DeJans Jr.
Lead Decision Scientist

July 15, 2025

Abstract

Linear and Mixed-Integer Programming (LP/MILP) has long served as the cornerstone of deterministic optimization in operations research. However, its traditional use of solving a one-shot, static problem is incompatible with the needs of modern systems that evolve over time under uncertainty. This paper reframes MILP not as a one-time solver, but as a policy class within the framework of *Sequential Decision Analytics* (SDA), using the language of *Cost Function Approximations* (CFAs) as defined by Warren Powell.

We show how to embed MILPs into decision-making pipelines that are executed repeatedly over time, where input data evolves, exogenous information arrives sequentially, and key cost or constraint parameters are tunable. These policies are expressive, auditable, and adaptable, especially when implemented using commercial solvers like Gurobi.

The article provides a rigorous treatment of MILP-as-policy, including mathematical foundations, parameter tuning strategies (offline and online), and deep domain-specific examples. It culminates in a complete, reproducible case study with implementation details.

This work is aimed at advanced practitioners and researchers who wish to operationalize optimization models not just for point solutions, but as living decision systems embedded in uncertain, dynamic environments.

Contents

1 From Optimization to Policy

1.1 The Legacy of Deterministic Optimization

For decades, linear and mixed-integer programming (LP/MILP) has served as the foundation of operations research. Problems such as workforce scheduling, inventory planning, production allocation, and routing are routinely formulated and solved as single-period optimization problems:

$$\min_{x \in \mathcal{X}} c^\top x \tag{1}$$

where $c \in \mathbb{R}^n$ is a vector of costs, and $\mathcal{X} \subseteq \mathbb{R}^n$ encodes all deterministic constraints. Once solved, the solution x^* is implemented directly.

This formulation assumes:

- All information is known upfront
- The system is stationary
- The solution will not need to adapt dynamically

These assumptions break down in most real-world settings.

1.2 The Reality of Sequential and Stochastic Systems

In modern applications (e.g., supply chains, transportation systems, real-time marketplaces) decisions must be made sequentially over time, in the face of evolving, uncertain information.

We define the sequential decision-making problem formally as follows:

At each time step $t = 0, 1, \ldots, T$, we observe a state $S_t \in \mathcal{S}$, and make a decision $x_t \in \mathcal{X}_t(S_t)$. After acting, we receive a contribution $C(S_t, x_t)$, observe new information W_{t+1}, and transition to a new state:

$$S_{t+1} = S^M(S_t, x_t, W_{t+1}) \tag{2}$$

The objective is to find a *policy* $\pi \in \Pi$, which maps states to actions $x_t^\pi(S_t)$, that maximizes the expected cumulative contribution:

$$\max_{\pi \in \Pi} \mathbb{E}^\pi \left[\sum_{t=0}^{T} C(S_t, x_t^\pi(S_t)) \right] \tag{3}$$

This is the **canonical objective of sequential decision analytics (SDA)**.

1.3 The Role of MILP in Dynamic Systems

Traditionally, MILPs are viewed as static, deterministic solvers. However, when embedded in a sequential loop, MILPs can act as *policy functions*. That is, the MILP structure defines how decisions are made, while certain components (such as cost vectors or constraint bounds) are treated as *tunable policy parameters*.

This leads us to the notion of **Cost Function Approximations (CFAs)**: one of the four foundational policy classes in SDA.

Key Idea: Instead of solving a deterministic MILP once, we solve a parameterized MILP repeatedly over time, treating it as a policy:

$$x_t^\pi(S_t) = \arg\min_{x \in \mathcal{X}_t(S_t)} c(x;\theta) \tag{4}$$

where θ are tunable parameters that govern the policy's behavior under uncertainty.

1.4 Motivating the CFA + MILP Approach

Why use MILP as a policy class?

- **Expressiveness**: MILPs can naturally express logical conditions, precedence, time windows, and network flows
- **Interpretability**: Solutions are explainable and auditable
- **Composability**: Easy to modify and scale across applications
- **Optimization Guarantees**: Solvers like Gurobi provide certificates of optimality and tolerances
- **Flexibility**: Policy tuning can reshape behavior without changing model structure

Yet doing this correctly requires *reframing* how MILP is modeled, solved, and tuned. This article shows how.

1.5 Article Roadmap

This article is structured as follows:

1. **Section 2** formalizes the SDA framework and clarifies the CFA class
2. **Section 3** explains how to embed MILPs into SDA
3. **Section 4** introduces offline and online tuning methodologies
4. **Section 5** presents canonical examples of CFA policy problems with MILP
5. **Section 6** delivers a fully-worked, end-to-end example in full technical detail

Each section includes mathematical formulations, modeling decisions, tuning strategies, and practical advice for implementation. The objective is to provide both theoretical clarity and computational guidance for practitioners designing decision policies via MILP.

---

2 The SDA Framework and Cost Function Approximations

2.1 The Universal Structure of Sequential Decision Problems

The Sequential Decision Analytics (SDA) framework, introduced in Powell's *Reinforcement Learning and Stochastic Optimization*, is a general-purpose, model-agnostic structure for representing any stochastic, dynamic decision problem.

At its core is the sequential decision model:

$$\mathcal{M} = (\mathcal{S}, \mathcal{X}, W, S^M, C) \tag{5}$$

where:

- $S_t \in \mathcal{S}$: The **state variable** at time t
- $x_t \in \mathcal{X}_t(S_t)$: The **decision variable**, feasible given the current state
- $W_{t+1} \in \mathcal{W}_{t+1}$: **Exogenous information**, arriving after the decision
- $S_{t+1} = S^M(S_t, x_t, W_{t+1})$: The **transition function** moving the system to its next state
- $C(S_t, x_t)$: The **contribution function**, measuring the value or cost of the decision

The goal is to compute a **policy** $\pi \in \Pi$, mapping states to decisions:

$$\pi : S_t \mapsto x_t^\pi(S_t) \tag{6}$$

that maximizes cumulative expected reward over a finite or infinite horizon:

$$\max_{\pi \in \Pi} \mathbb{E}^\pi \left[\sum_{t=0}^{T} C(S_t, x_t^\pi(S_t)) \right] \tag{7}$$

This framework is expressive enough to cover:

- Dynamic programming
- Stochastic optimization
- Reinforcement learning
- Simulation-based optimization
- Real-time control systems

2.2 A Word on State Variables

The **state variable** S_t is a first-class citizen in SDA. It must be *sufficient*: it should contain all the information necessary to make an optimal decision and to simulate the system forward.

Typical elements of S_t might include:

- Inventory levels
- Time-dependent forecasts

- Machine availability
- Order backlog
- Vehicle locations
- Rolling averages of historical quantities

Practitioners often under-specify S_t, leading to poor policies. In CFA modeling, S_t provides the **data inputs** for MILP construction at time t.

2.3 Policy Classes in SDA

The power of SDA lies in its **unifying view of policies**. It does not dictate a specific solution method, but rather classifies the kind of policy function you're using. Powell defines four canonical policy classes:

Policy Function Approximations (PFA): These map states directly to actions:

$$x_t^\pi(S_t) = \pi(S_t; \theta) \tag{8}$$

PFAs include:

- Heuristics
- Rule-based systems
- Neural networks
- Parametric functions

Pros: Fast to evaluate
Cons: Hard to tune under complex constraints

Value Function Approximations (VFA): These learn the value of being in a state:

$$V(S_t; \theta) \approx \mathbb{E}\left[\sum_{t'=t}^{T} C(S_{t'}, x_{t'}^\pi(S_{t'}))\right] \tag{9}$$

Then select decisions via:

$$x_t^\pi(S_t) = \arg\max_{x \in \mathcal{X}_t(S_t)} \left[C(S_t, x) + \mathbb{E}[V(S_{t+1})]\right] \tag{10}$$

VFAs are at the heart of:

- Q-learning
- Approximate dynamic programming
- Bellman-based reinforcement learning

Pros: Theoretically grounded
Cons: Hard to construct in high-dimensional or discrete-action spaces

Direct Lookahead Approximations (DLA): DLAs simulate future information and solve the lookahead problem:

$$x_t^\pi(S_t) = \arg\max_{x \in \mathcal{X}_t(S_t)} \mathbb{E}\left[\sum_{t'=t}^{t+H} C(S_{t'}, x_{t'})\right] \tag{11}$$

This requires sampling future exogenous information $W_{t+1}, \ldots, W_{t+H}$.

Pros: Captures complex uncertainty
Cons: Computationally expensive

Cost Function Approximations (CFA), Our Focus: CFAs define a parametric optimization model (e.g., MILP) as a decision policy:

$$x_t^\pi(S_t) = \arg\min_{x \in \mathcal{X}_t(S_t)} \left[c(x; \theta) + \sum_{i=1}^{m} \lambda_i f_i(x, S_t)\right] \tag{12}$$

Where:

- $c(x; \theta)$: Parametric cost function
- $f_i(x, S_t)$: Penalty terms (e.g., slack, soft constraints)
- λ_i: Tunable weights

Key Insight: The *policy* is defined by solving the optimization model, and *tuning the parameters* θ defines how it behaves under uncertainty.

MILPs become policies through CFAs.

2.4 Why CFAs Are Ideal for MILP

Feature	**Explanation**
Structure	CFAs retain the integrity of constraints and combinatorics
Interpretability	Each constraint and penalty is semantically meaningful
Parametric Tuning	Parameters act as knobs that guide behavior
Solver Compatibility	Gurobi is well-suited for repeated, warm-started solves
Deployment Ready	MILP policies are auditable, explainable, and compliant

Table 1: Why CFA policies are ideal for MILP in sequential decision settings

2.5 Summary

The SDA framework models time, information, and uncertainty explicitly. Within this structure, CFAs offer a principled way to use MILP as a policy. The structure remains fixed; the state and parameters change. This opens the door to adaptive, real-time, data-driven decision systems using a familiar and auditable optimization toolchain.

In the next section, we'll operationalize this idea: how to construct and use MILPs as CFA policies within SDA.

3 Embedding MILP into SDA as a Cost Function Approximation

3.1 MILP as a Policy: A Fundamental Reframing

In classical deterministic optimization, a Mixed-Integer Linear Program (MILP) is used to compute a one-time decision:

$$x^* = \arg\min_{x \in \mathcal{X}} c^\top x \tag{13}$$

This is **not** a policy. It is a point solution. There is no state, no time, no uncertainty, no learning.

To use MILP as a *policy* within the SDA framework, we must reformulate it as a parameterized function that maps each state S_t to a decision x_t, by solving a model that *does not change structure*, but receives new state-dependent data and adjustable policy parameters at each time step:

$$x_t^\pi(S_t; \theta) = \arg\min_{x \in \mathcal{X}_t(S_t)} c^\top x + \sum_{i=1}^{m} \lambda_i f_i(x, S_t) \tag{14}$$

This is the **defining equation of a CFA**. The policy is implemented by solving a mathematical program (MILP), where:

- $\mathcal{X}_t(S_t)$: The feasible set, constructed dynamically based on the current state
- c, λ: Cost vector and penalty weights
- f_i: Penalty functions, often linear or piecewise linear

3.2 Policy Inputs: Decoupling Data and Parameters

Each time the policy is executed, the MILP receives two types of inputs:

1. State-dependent data S_t These are derived from the current environment and include:

- Forecasts (e.g., demand, travel times)
- Physical system state (e.g., inventory, capacity, position)
- Environmental/regulatory signals
- Rolling aggregates or lagged statistics

These data enter the model as coefficients, right-hand sides, or constraint bounds and are considered *exogenous.* They should **not** be tuned.

2. Policy parameters θ These parameters govern the policy's behavior and are subject to tuning. Examples include:

- Weights in multi-objective tradeoffs (e.g., cost vs. service)
- Penalty coefficients for soft constraint violations
- Virtual buffers (e.g., slack multipliers, lead time inflation)
- Priority rules encoded via weighted penalties

These are not statistically learned, but instead **tuned**, offline via simulation, or online via feedback.

3.3 Structure of the MILP

The CFA formulation relies on the MILP's *fixed structure*. The decision variables, constraints, and logic (e.g., precedence, timing, logical conditions) remain invariant across time. The state variable S_t influences only the coefficients and right-hand sides.

A general CFA MILP takes the form:

$$\begin{aligned} \min_{x,z} \quad & c^\top x + \sum_{i=1}^{m} \lambda_i f_i(x, z; S_t) \\ \text{s.t.} \quad & Ax + Bz \le b(S_t) \\ & x \in \mathbb{R}^n_+, \quad z \in \{0,1\}^p \end{aligned} \tag{15}$$

Where:

- x: Continuous decision variables
- z: Binary/integer decision variables
- $b(S_t)$: State-dependent constraint bounds
- $f_i(x, z; S_t)$: Penalty functions or surrogate losses

Tuning affects λ_i, not the formulation. Adaptivity arises via updated S_t.

3.4 Algorithmic Flow

To deploy a CFA policy, the implementation pipeline is as follows:

Offline Stage (Design & Tuning)

1. Define the sequential decision problem, state S_t, and exogenous process W_{t+1}
2. Construct a fixed-structure MILP, parameterized by θ
3. Develop the $S_t \rightarrow$ MILP mapping (coefficients, bounds, RHS)
4. Select initial tuning parameters θ
5. Run simulations to evaluate and improve the policy

Online Stage (Execution Loop)

1. Observe or compute state S_t
2. Instantiate MILP model with S_t and θ
3. Solve MILP using Gurobi
4. Apply resulting decision x_t
5. Observe W_{t+1} and compute next state: $S_{t+1} = S^M(S_t, x_t, W_{t+1})$

This loop is repeated for $t = 0, 1, \dots, T$.

3.5 Practical Implementation Guidance

Software Structure

A modular, object-oriented implementation should separate:

- **Model definition**: Fixed structure (variables, constraints, objective form)
- **State-to-data mapping**: How S_t populates coefficients, bounds, RHS
- **Parameter injection**: How θ modifies weights and tolerances
- **Solver configuration**: Gurobi parameters (presolve, tolerances, MIP start)
- **Simulation logic**: Rolling horizon loop, state transitions

Warm-starting

MILPs solved repeatedly can be accelerated via warm starts:

- Reuse prior variable values via Gurobi's `Start` attribute
- Cache solver state for minimal model regeneration
- Apply basis information (for LP relaxations)

```
Python (Gurobi):
model.setAttr(GRB.Attr.Start, var, previous_solution[var])
```

Rolling Horizon and Lookahead

To mitigate myopic decisions, CFA policies can incorporate a lookahead window of H steps:

$$x_t^\pi(S_t) = \arg\min_{x_{t:t+H}} \sum_{h=0}^{H} \gamma^h C(S_{t+h}, x_{t+h}; \theta) \tag{16}$$

This formulation approximates value over a planning horizon and improves robustness to short-term noise.

3.6 Summary

By embedding MILPs within the SDA framework as Cost Function Approximations, we move from "solving a problem once" to "solving a parameterized policy repeatedly." This paradigm shift transforms MILP from a static optimizer into a decision engine under uncertainty.

The MILP structure is fixed, but the decision process is dynamic. The policy is adaptive, interpretable, and deployable.

In the next section, we address how to tune these policies effectively; both offline in simulation and online using feedback.

---

4 Tuning CFA Policies: Offline and Online Techniques

4.1 The Nature of Tuning in CFA Policies

Unlike statistical models trained via supervised learning, CFA policies are **tuned**, because they involve decision-making under uncertainty, not prediction.

Tuning in CFAs refers to choosing the best parameter vector θ (e.g., penalty weights, target levels, tradeoff coefficients) to shape the behavior of a policy:

$$x_t^\pi(S_t;\theta) = \arg\min_{x \in \mathcal{X}_t(S_t)} c(x;\theta) \tag{17}$$

Here, the MILP structure is fixed, and performance is evaluated by simulating full sample trajectories of the exogenous process $W_{t+1} \sim \mathbb{P}_{t+1}$. There is no closed-form objective for θ, making this a **simulation-based, black-box optimization problem**.

4.2 Formal Statement of the Tuning Problem

Let:

- $\Theta \subseteq \mathbb{R}^d$: Feasible space of parameter vectors
- $\omega \in \Omega$: A trajectory of exogenous information
- π^θ: The CFA policy defined by θ
- $F^\pi(\omega;\theta)$: Total cost incurred by π^θ over trajectory ω

The tuning problem becomes:

$$\min_{\theta \in \Theta} \mathbb{E}_{\omega \sim \Omega} \left[F^\pi(\omega;\theta) \right] \tag{18}$$

Each evaluation of θ requires solving many MILPs across time. This is a high-cost, stochastic optimization problem.

4.3 Offline Tuning Methodologies

4.3.1 Grid Search

- Define a grid over low-dimensional θ space
- Simulate rollout for each θ
- Choose best performing vector

Pros: Simple, exhaustive; **Cons:** Exponential cost, poor scaling

```
for lambda_b in [10, 50, 100, 500]:
    avg_cost = simulate_rollout(lambda_b)
```

4.3.2 Random Search

Randomly sample $\theta_1, \ldots, \theta_k \sim \text{Uniform}(\Theta)$:

- Roll out each π^{θ_i}
- Record average cost
- Select best performing θ

Pros: Scales better, simple to parallelize
Cons: Inefficient in small search spaces

4.3.3 Bayesian Optimization

Model $F^{\pi}(\theta)$ using a surrogate function (e.g., GP, RF) and optimize an acquisition function to choose θ.

- Sample-efficient tuning
- Supports noisy, expensive evaluations

Use libraries such as `BoTorch`, `Optuna`, or `BayesOpt`.

4.3.4 Evolutionary Search

Techniques such as CMA-ES, DE, or ES evolve a population of parameter vectors:

- Evaluate population across simulations
- Apply mutation, crossover, elitism

Pros: Parallelizable, flexible
Cons: Requires large number of evaluations

4.3.5 Hypergradient-Based Tuning

If the optimization model is differentiable, apply:

$$\frac{\partial F^{\pi}(\omega; \theta)}{\partial \theta} \tag{19}$$

Feasible for LPs and differentiable surrogates. Difficult for MILPs due to discrete nature.

4.4 Design of Simulation Environments

When tuning CFA policies, simulation fidelity is essential:

Feature	**Description**
Random Seed Control	Ensures experiment repeatability
Rolling Horizon Logic	Match how the policy will run online
Exogenous Process Simulation	Simulate $W_{t+1} \sim \mathbb{P}_{t+1}$
State Transition Accuracy	Implement $S_{t+1} = S^M(S_t, x_t, W_{t+1})$
Metric Logging	Track KPIs like cost, service, feasibility

4.5 Online Tuning: Feedback-Based Adaptation

Environments may shift over time. Online tuning allows θ to be adjusted in real time.

4.5.1 Reactive Rule-Based Updates

Track a KPI K_t and adjust parameter λ_t via:

$$\lambda_{t+1} = \lambda_t + \eta(K^* - K_t) \tag{20}$$

- η: Learning rate
- K_t: Observed performance metric

Pros: Easy to implement
Cons: Sensitive to tuning, no guarantees

4.5.2 Multi-Armed Bandit (MAB) Tuning

Discretize Θ into k arms $\theta^{(1)}, \dots, \theta^{(k)}$. Use explore-exploit methods (e.g., UCB, Thompson Sampling):

- Select arm at each step
- Observe reward
- Update beliefs

Pros: Theoretically grounded, efficient
Cons: Requires discrete tuning space

4.5.3 Meta-MILP Tuning

Model the choice of θ as a decision within an outer MILP:

- Introduce binary variables for each θ_i
- Encode selection constraints
- Solve outer model jointly with MILP policy

Pros: Full integration into MILP stack
Cons: High computational burden

4.5.4 RL-Augmented Parameter Tuning

View θ_t as a meta-decision in a higher-level MDP:

- Train a reinforcement learning agent over parameter space
- Use rollout feedback to update policy

Pros: Adapts to nonstationarity
Cons: Complex and data-hungry

4.6 Evaluation Metrics

Align metrics with business outcomes:

Metric	Description
Total Cost	Sum of economic penalties, direct losses
Service Level	Percent of demand fulfilled on time
Fairness	Equitable treatment across requests or users
Robustness	Variance or worst-case performance
Latency	Average solve time per decision epoch

Composite metrics often reflect stakeholder objectives better than raw solver costs.

4.7 Summary

Tuning is where CFA policies become powerful decision-making tools:

- **Offline tuning** shapes initial performance
- **Online tuning** allows ongoing adaptation
- **Well-chosen metrics** ensure business alignment

Takeaway: A CFA policy is not just a model, it is a living, adaptive mechanism for operational intelligence.

---

5 CFA Domain Examples with MILP (High-Level)

This section presents three canonical examples of CFA policy design using MILP within the SDA framework. Each case follows the standard SDA modeling structure:

- **State variable** S_t
- **Decision variable** x_t
- **Exogenous information** W_{t+1}
- **Transition function** $S_{t+1} = S^M(S_t, x_t, W_{t+1})$
- **Contribution function** $C(S_t, x_t)$

- **MILP formulation** as a CFA
- **Tunable parameters** θ
- **Tuning and implementation guidance**

5.1 Example 1: Inventory Replenishment with Backorders

Context: A warehouse must decide how much to reorder each day to meet stochastic customer demand D_t. Orders arrive with a fixed lead time of one day. If demand exceeds available inventory, unmet demand becomes *backlogged* and is carried into future periods with an associated penalty. Excess inventory incurs holding cost.

SDA Elements:

- $S_t = (I_t, B_t, \hat{D}_{t+1:t+H})$: On-hand inventory, backlog, and forecasted demand over a rolling horizon.
- $x_t \in \mathbb{R}_+$: Quantity to order at time t (delivered at $t+1$).
- $W_{t+1} = D_t \sim \mathbb{P}_t$: Realized demand at t.
- $S_{t+1} = (I_{t+1}, B_{t+1}, \hat{D}_{t+2:t+H+1})$: Inventory and backlog evolve deterministically given x_t and D_t.
- $C(S_t, x_t) = -c_o x_t - c_h(I_t + x_t - D_t)^+ - \lambda_b(D_t - I_t - x_t)^+$: Penalizes order cost, holding cost, and backlog.

MILP Formulation (1-period horizon):

$$\begin{aligned}
\min_{x_t, y_t, b_t} \quad & c_o x_t + c_h(I_t + x_t - y_t) + \lambda_b b_t \\
\text{s.t.} \quad & y_t + b_t = \hat{D}_t + B_t \quad \text{(demand balance)} \\
& y_t \leq I_t + x_t \quad \text{(inventory constraint)} \\
& x_t, y_t, b_t \geq 0
\end{aligned}$$

Definitions:

- y_t: Quantity of demand fulfilled at time t.
- b_t: Remaining backorder after t.
- $\hat{D}_t$: Forecasted demand for time t.
- B_t: Unfulfilled backlog carried from $t-1$.

Tunable Parameters: λ_b (backlog penalty), c_h (holding cost), forecast horizon H

Guidance:

- Solve at each time t using updated $\hat{D}_t$.
- Use warm starts across rolling horizons to improve solve time.
- Tune λ_b to trade off service level vs. cost.

5.2 Example 2: Vehicle Routing with Time Windows (VRPTW)

Context: A fleet of delivery vehicles must fulfill customer requests within time windows. New requests arrive over time, and travel times may vary due to traffic. Late arrivals incur penalty. The goal is to minimize total travel cost and lateness.

SDA Elements:

- $S_t = (\mathcal{R}_t, \mathcal{V}_t)$: Current delivery requests and vehicle states.
- x_t: Routing decision (binary selection of arcs between stops).
- W_{t+1}: Exogenous changes such as new requests or traffic delays.
- S_{t+1}: Updated route queue and vehicle availability.
- $C(S_t, x_t) = -\sum_{(i,j)} c_{ij} x_{ij} - \lambda_L \sum_i \text{Late}_i(x_t)$

MILP Formulation:

- Variables:
  - $x_{ij}^v \in \{0, 1\}$: Whether vehicle v travels from i to j.
  - T_i: Time of service at customer i.

$$\min_{x,T} \quad \sum_{(i,j),v} c_{ij} x_{ij}^v + \lambda_L \sum_i \max(0, T_i - \tau_i^{\text{due}})$$

Subject to:

- Flow constraints to ensure valid vehicle paths.
- Time propagation along arcs.
- Service windows: $T_i \in [\tau_i^{\text{earliest}}, \tau_i^{\text{due}}]$
- Capacity constraints on each vehicle.

Tunable Parameters: λ_L (lateness penalty), c_{ij} (arc cost), service time buffers

Guidance:

- Use Gurobi callbacks for subtour elimination.
- Update c_{ij} dynamically with real-time traffic or fuel prices.
- Offline tune λ_L to balance delivery punctuality with cost.

5.3 Example 3: Dealer Supply Allocation with Preferences

Context: A central planner must allocate limited supply of finished vehicles across dealerships. Each dealer has stated vehicle preferences. The planner seeks to maximize utility while maintaining fairness and minimizing preference deviation.

SDA Elements:

- $S_t = (\mathcal{D}_t, \mathcal{S}_t)$: Dealer demand forecasts and current supply.

- $x_t \in \mathbb{Z}_+^{n_d \times n_v}$: Allocation decision of vehicles to dealers.
- W_{t+1}: Updates to demand or production disruptions.
- S_{t+1}: Updated state of vehicle supply and dealer expectations.
- $C(S_t, x_t) = \sum_{d,v} \text{Utility}_{dv} x_{dv} - \lambda_R \cdot \text{VarianceAcrossRegions}(x_t)$

MILP Formulation:

$$\begin{aligned}
\max_x \quad & \sum_{d,v} \text{Utility}_{dv} x_{dv} - \lambda_R \cdot \text{RegionalImbalance}(x) - \lambda_U \sum_{d,v} |x_{dv} - \text{Pref}_{dv}| \\
\text{s.t.} \quad & \sum_d x_{dv} \le \text{Supply}_v \quad \forall v \\
& \sum_v x_{dv} \le \text{MaxDemand}_d \quad \forall d \\
& x_{dv} \in \mathbb{Z}_+
\end{aligned}$$

Tunable Parameters: λ_U (preference penalty), λ_R (regional fairness), prioritization weights

Guidance:

- Model preference as soft constraints with penalized deviation.
- Use fairness metrics (e.g., regional Gini coefficient) in RegionalImbalance.
- Encourage sparsity in x for realistic assignment patterns.

Summary Table of CFA Modeling Elements

Example	**State** S_t	**Decision** x_t	**Contribution** $C(S_t, x_t)$	**Key Parameters** θ
Inventory	Inventory, backlog, forecast	Order quantity	Cost of ordering, holding, backlogging	λ_b, c_h
VRPTW	Delivery queue, vehicle state	Route selection	Travel + lateness cost	λ_L, c_{ij}
Allocation	Supply, demand, preferences	Vehicle assignment	Utility − fairness − preference gap	λ_U, λ_R

---

6 Implementing a CFA Policy with Gurobi (Low-Level)

This section synthesizes all prior concepts into a fully worked CFA implementation for a sequential inventory replenishment problem. The approach includes model definition, MILP formulation, simulation environment, tuning methodology, Gurobi integration, and deployment guidance. Additional diagnostic and performance techniques are provided to support practitioners in rigorous implementation.

6.1 Problem Overview: Inventory Replenishment with Stochastic Demand and Backorders

A warehouse faces daily stochastic demand and must decide how much inventory to order. Unsatisfied demand incurs backorder penalties; excess inventory incurs holding costs. Replenishment arrives with one-day lead time. A CFA policy is used, with a rolling MILP solved via Gurobi at each time step.

The objective is to minimize long-run average cost while achieving high service levels.

6.2 SDA Model Specification

State Variable:

$$S_t = (I_t, B_t, \hat{D}_{t+1:t+H})$$

- $I_t \in \mathbb{Z}_+$: On-hand inventory at time t
- $B_t \in \mathbb{Z}_+$: Backlog of unfulfilled demand from prior periods
- $\hat{D}_{t+1:t+H} \in \mathbb{R}^H$: Forecast of demand over a rolling horizon H

Decision Variable:

- $x_t \in \mathbb{R}_+$: Quantity to order, arrives next day

Exogenous Information:

- $D_t \sim \mathbb{P}_t$: Realized stochastic demand

Transition Function:

$$\begin{aligned} y_t &= \min(I_t + x_t, D_t + B_t) \\ I_{t+1} &= I_t + x_t - y_t \\ B_{t+1} &= D_t + B_t - y_t \\ S_{t+1} &= (I_{t+1}, B_{t+1}, \hat{D}_{t+2:t+H+1}) \end{aligned}$$

Contribution Function:

$$C(S_t, x_t) = -\left[c_o x_t + c_h I_{t+1} + \lambda_b B_{t+1}\right]$$

Parameters:

- c_o: Unit cost to order
- c_h: Per-unit inventory holding cost
- λ_b: Backorder penalty per unit short

Typical use cases: e-commerce fulfillment, spare part restocking, daily store replenishment.

6.3 Rolling-Horizon MILP Formulation (One-Step Lookahead)

The CFA policy solves this MILP daily using current forecasts and backlog:

Decision Variables:

- $x \in \mathbb{R}_+$: Quantity ordered
- $y \in \mathbb{R}_+$: Fulfilled demand
- $b \in \mathbb{R}_+$: Backorder quantity

Objective:

$$\min_{x,y,b} \quad c_o x + c_h(I_t + x - y) + \lambda_b b$$

Subject to:

$$\begin{aligned} y + b &= \hat{D}_t + B_t \quad \text{(demand balance)} \\ y &\leq I_t + x \quad \text{(inventory constraint)} \\ x, y, b &\geq 0 \end{aligned}$$

This formulation captures the tradeoff between ordering cost, holding inventory, and backordering demand.

Extension for Horizon $H > 1$: Practitioners may roll out this MILP over multiple future days (e.g., $H = 3$) using forecasted demand and recursive inventory equations, though dimensionality increases.

Numerical Example Parameters:

- $c_o = 5$, $c_h = 0.5$, $\lambda_b = 20$
- $I_0 = 30$, $B_0 = 0$, $\mu_t = 20$ (Poisson mean)
- Forecast $\hat{D}_{t+1:t+3} = [21, 19, 20]$

6.4 Simulation Environment Setup

To validate and tune CFA policies using Gurobi in a realistic setting, we construct a simulation environment that reflects daily operational dynamics under uncertainty. The goal is to evaluate the long-term performance of the policy under a stochastic sequence of events using multiple rollouts.
Inputs and Assumptions

- **Planning horizon:** $T = 365$ days
- **Demand model:** $D_t \sim \text{Poisson}(\mu_t)$ with μ_t potentially varying with seasonality, weekday effects, or promotional campaigns
- **Forecast model:** $\hat{D}_t = \mathbb{E}[D_t] + \varepsilon_t$ where ε_t captures forecast noise
- **Forecast horizon:** $H = 1$, but framework extends naturally to $H > 1$ with rolling demand vectors
- **Initial state:** $I_0 = 50$, $B_0 = 0$
- **Fixed costs:** c_o, c_h, and λ_b must be set or explored via tuning

Forecasting Considerations To simulate realistic operations:

- Forecast $\hat{D}_t$ is updated at each t using a trained model (e.g., ARIMA, moving average, or ML-based regressor)
- Forecast noise ε_t may be Gaussian ($\mathcal{N}(0, \sigma^2)$) or drawn from historical forecast errors
- Scenario testing can inject structured noise: e.g., biased forecasts during holidays

Simulation Loop: Each simulation episode mimics a full operational year:

```python
for episode in range(num_simulations):
    I, B = I_0, B_0
    history = []
    for t in range(T):
        forecast = demand_model.predict(t) + forecast_noise(t)
        S_t = (I, B, [forecast])
        x_t = solve_milp(S_t, lambda_b)
        D_t = sample_demand(t)
        y_t = min(I + x_t, D_t + B)
        I = I + x_t - y_t
        B = D_t + B - y_t
        cost = c_o * x_t + c_h * I + lambda_b * B
        history.append((t, x_t, I, B, D_t, cost))
        log_episode_metrics(history)
```

Diagnostics and Enhancements

- **Warm start carryover:** Cache previous MILP solutions for faster solves
- **Outlier tracking:** Detect weeks with high backlogs or large costs to refine λ_b
- **Rolling forecast window:** Store $\hat{D}_{t:t+H-1}$ instead of just $\hat{D}_t$
- **State-action recording:** Store $(S_t, x_t, D_t, I_t, B_t, C_t)$ tuples to support offline policy learning or supervised approximators
- **Multiple demand scenarios:** Run sensitivity analysis across various demand seasonality and shock regimes (e.g., promotions, stockouts)

Performance Metrics Captured Per Episode

- Average daily cost
- 95th percentile backlog
- Cumulative fill rate
- Mean time between backorders
- Inventory turnover ratio

6.5 MILP Implementation with Gurobi

This section shows how to compile the CFA objective into a MILP using the Gurobi Python API, solved repeatedly within a simulation or live system.

Objective Structure:

The rolling-horizon MILP is derived from the post-decision state value approximation:

$$\min_{x \in \mathbb{R}_+} \quad c_o x + \min_{y,b} \left[c_h (I_t + x - y) + \lambda_b b \right]$$

The inner minimization handles demand balancing, while the outer variable x is the replenishment control.

Implementation:

```python
def solve_milp(S_t, lambda_b):
    I_t, B_t, forecast = S_t
    D_hat = forecast[0]

    model = gp.Model("CFA_Inventory")
    model.Params.OutputFlag = 0  # silence solver output

    # Variables
    x = model.addVar(lb=0, name="order")
    y = model.addVar(lb=0, name="fulfilled")
    b = model.addVar(lb=0, name="backlog")

    # Constraints
    model.addConstr(y + b == D_hat + B_t, name="demand_balance")
    model.addConstr(y <= I_t + x, name="inventory_limit")

    # Objective
    holding_cost = c_h * (I_t + x - y)
    penalty_cost = lambda_b * b
    ordering_cost = c_o * x
    model.setObjective(ordering_cost + holding_cost + penalty_cost, GRB.
    MINIMIZE)

    # Solve and return
    model.optimize()
    return x.X
```

Tips for Robustness and Speed:

- Reuse the model instance across time steps using `model.reset()`
- Use `model.update()` carefully to replace objective coefficients
- When tuning, store dual variables and slack to detect policy stress
- Set time or MIPGap limits in high-throughput environments
- Use Gurobi Model Pooling to evaluate multiple near-optimal solutions if needed

6.6 Offline Tuning Procedure

Offline tuning is used to optimize the penalty parameter λ_b by simulating the CFA policy under fixed parameters.

```python
def tune_lambda_b(lambda_grid):
    best_lambda = None
    best_cost = float('inf')
    for lambda_b in lambda_grid:
        avg_cost = simulate_policy(lambda_b)
        print(f"  ={lambda_b}     cost={avg_cost:.2f}")
        if avg_cost < best_cost:
            best_cost = avg_cost
            best_lambda = lambda_b
    return best_lambda
```

Lambda Sweep Example:

$$\lambda_b \in \{1, 5, 10, 20, 50, 100, 200, 500\}$$

Guidance:

- Perform 100+ replications per λ_b value to smooth cost variation
- Log fill rate, backlog frequency, and service levels
- Plot cost(λ_b) and identify the Pareto frontier
- Optionally fit a local polynomial to interpolate and find the minimum

Plot Results:

- Total cost vs λ_b
- Service level vs λ_b

Select λ_b that minimizes cost while meeting target service level.

6.7 Online Adaptation Strategy

To adapt to nonstationary demand or policy drift, update the penalty parameter λ_b dynamically based on observed performance:

$$\lambda_{b,t+1} = \lambda_{b,t} + \eta(s^* - s_t)$$

- s_t: empirical service level observed over a rolling window
- s^*: desired fill rate (e.g., 0.98)
- η: learning rate (e.g., 0.1 or use RMSprop variant)

Implementation Note:

- You may choose to project λ_b back into a safe range: $\lambda_b \in [0, 500]$
- Store history of λ_b updates and cost metrics to evaluate convergence

6.8 Evaluation Metrics

In practice, a single scalar cost may obscure policy behavior. Use the following set of performance diagnostics:

Metric	**Formula**
Total cost	$\sum_{t=1}^{T} (c_o x_t + c_h I_t + \lambda_b B_t)$
Fill rate	$1 - \frac{\sum_{t=1}^{T} B_t}{\sum_{t=1}^{T} D_t}$
Backlog frequency	$\frac{\#\{t : B_t > 0\}}{T}$
Inventory turnover	$\frac{\sum_t D_t}{\frac{1}{T}\sum_t I_t}$
Stockout events	$\#\{t : I_t = 0 \text{ and } B_t > 0\}$

Visualization Tips:

- Plot cumulative cost over time to observe ramp-up behavior
- Scatterplot: Cost vs Fill rate across tuning grid
- Use violin plots or boxplots across simulation replications

6.9 Production Deployment Tips

Bridging simulation to production involves thoughtful system design and operational safeguards. Key strategies include:

- **Encapsulate the MILP solver in a stateless microservice**: Expose a REST interface that accepts state S_t and returns the computed decision x_t. Ensure compatibility with upstream forecasters and downstream execution systems.
- **Leverage warm starts and persistent models**: If the structure of the MILP remains fixed across time steps (as in most CFA applications), retain the Gurobi model object in memory and update only coefficients (e.g., D_t, I_t, B_t). This significantly reduces solve time.
- **Enable Gurobi Model Pools**: In mission-critical settings, use the solution pool to evaluate multiple near-optimal decisions and inject robustness or diversity into your selection policy.
- **Cache decisions for repeated states**: In high-throughput environments with repetitive states (e.g., low-dimensional S_t), memoization can eliminate redundant solves entirely.
- **Guardrail constraints**: Add sanity checks such as:
  - Maximum/minimum allowed order size ($x_t \leq x_{\max}$)
  - Holding capacity ($I_t + x_t \leq I_{\max}$)
  - Constraint violations due to bad forecasts
- **Track online KPIs and enable adaptive tuning**: Log service level, cost, backlog trends, and solver diagnostics. Use this data to periodically re-tune λ_b, detect drift, or trigger fallback logic (e.g., myopic policy if MILP fails).
- **Validate end-to-end integration**: Confirm that the forecast system, state encoder, MILP solver, and inventory execution system (ERP/WMS) all communicate with consistent semantics and units.

Additional Considerations:

- Containerize with Docker for portability
- Use async job queues for high-latency solves
- Monitor solver logs to detect infeasibilities or timeouts
- Implement retry and fallback policies
- Use staged rollout (canary deployments) before full production cutover

6.10 Summary of Implementation Steps

To summarize, here is the step-by-step process to build a full CFA-based inventory policy using Gurobi:

Step	Action
1	Define the SDA model: S_t, x_t, W_{t+1}, and transition model S^M
2	Formulate the rolling-horizon MILP with tunable parameter λ_b
3	Implement the MILP solver in Python with Gurobi integration
4	Build a simulation environment that generates demand, tracks state transitions, and evaluates cost
5	Perform offline tuning of λ_b to identify optimal penalty settings
6	(Optional) Add online adaptation using service-level feedback
7	Evaluate the policy with robust metrics: cost, service level, backlog frequency
8	Wrap the CFA policy in a scalable microservice architecture
9	Monitor KPIs in production and maintain retraining/tuning pipeline

Conclusion

This case study has demonstrated how to construct a fully operational CFA policy using Gurobi to solve a sequential inventory control problem under uncertainty. Each phase (from model formulation, simulation, and offline tuning to real-time deployment) has been decomposed and illustrated with reproducible code and implementation guidance.

More broadly, this example illustrates how Gurobi-based MILPs can be embedded within Sequential Decision Analytics frameworks to solve complex, dynamic, and uncertain real-world problems. The CFA approach, with its tunable cost approximations and modular design, provides a flexible blueprint for extending deterministic optimization into uncertain, evolving systems. Practitioners are encouraged to experiment with more complex state variables, multi-stage lookaheads, or joint decisions across resources to scale this pattern to broader operations research challenges.

---

7 Conclusion

This article has shown how to reframe traditional MILP modeling from a static optimization exercise into a dynamic decision policy framework using Cost Function Approximations (CFAs). By embedding MILPs inside the Sequential Decision Analytics (SDA) architecture, we unlock a powerful way to make real-time, uncertainty-aware decisions; without abandoning the rigor, transparency, or structure that MILPs offer.

Far from being obsolete in a world of machine learning and black-box AI, MILPs remain one of the most interpretable and controllable tools in the decision scientist's toolkit. When combined with simulation-based tuning, rolling-horizon deployment, and online adaptation, they serve not only as solvers, but as policies that can evolve with the system they operate in.

A MILP is no longer just a one-time plan. It is a living, tunable, operational artifact that governs decisions under uncertainty.

By embracing this perspective, we move from solving problems on paper to designing policies that shape behavior in the real world. As the fields of operations research, simulation, and AI converge, decision scientists equipped with tools like Gurobi, and the mindset of sequential design, will lead the way.