

Automated Design of Inventory Policy with Large Language Models: An Exploratory Study

Fenghua Yang, Preet Baxi, Stefanus Jasin

University of Michigan, Ann Arbor, MI 48109, USA (yfenghua@umich.edu, preetb@umich.edu, sjasin@umich.edu)

Yi Zhang

Stanford University, Stanford, CA 94305, USA (yz5195@stanford.edu)

Yanzhe Lei

Queen’s University, Kingston, ON K7L 3N6, Canada (yl64@queensu.ca)

Mo Liu

University of North Carolina, Chapel Hill, NC 27599, USA (mo_liu@unc.edu)

Parshan Pakiman

University at Buffalo, State University of New York, Buffalo, NY 14260, USA (parshanp@buffalo.edu)

Firms making inventory decisions increasingly have access to rich operational data, advanced optimization tools, and rapidly improving large language models (LLMs). Typically, data characterize the operating environment, optimization selects parameter values within a prespecified inventory policy class, and LLMs support tasks such as coding and decision analysis. We develop an integrated framework that combines these resources to automate inventory policy design. Given demand data, the framework iteratively uses an LLM to generate parameterized policy classes and an external solver to optimize its parameters within each class. Across 30 lost-sales inventory instances, the mean cost reduction relative to optimized base-stock benchmarks increases from 17.5% after one generation to 30.0% after ten generations. Parameter optimization is central to this performance: an LLM-only variant performs substantially worse, whereas optimization-guided feedback improves policy quality, accelerates search, and directs the LLM toward better policy classes rather than merely better parameter values within a fixed class. The strongest discovered policies are also interpretable: they combine recognizable inventory-control motifs, including capped orders, discounted or weighted pipeline inventory, and threshold-based replenishment logic. The search thereby produces new policy-class functional forms that, to our knowledge, have not previously been studied in the lost-sales inventory literature. These functional forms are not specified *ex ante* but emerge from the search process. Moreover, after their parameters are re-optimized, three discovered policy classes achieve average cost reductions of 21.75% – 22.60% across 10,064 new inventory instances. Overall, the results show that data-driven parameter optimization can guide LLM-based search over a broad space of inventory policy classes and identify high-performing, interpretable, and transferable decision rules.

1. Introduction

Optimization models have long formed the backbone of inventory decision making, providing principled methods for characterizing, computing, and improving replenishment policies within specified policy classes (Arrow et al. 1951, Scarf 1960, Veinott and Wagner 1965, Zipkin 2000, Porteus 2002). The growth of operational data has enabled firms to complement these models with richer information, including sales histories, inventory records, and contextual features, to improve demand forecasting, demand estimation, and replenishment decisions (Iyer and Schrage 1992, Fildes et al.

2009, DeHoratius et al. 2008, Huh et al. 2011, Ding et al. 2024). More recently, large language models (LLMs) and generative artificial intelligence have emerged as a third resource, with growing potential to support optimization practice, supply-chain management, and broader managerial decision making (Jiang et al. 2026, Chen et al. 2026, Csaszar et al. 2024, Cohen et al. 2026, McKinsey & Company 2025). Thus, an important question is not simply whether LLMs will be used in practice, but how they can be used effectively. In inventory control, these three resources are typically assigned distinct roles: data support forecasting and demand estimation, optimization improves decisions within prespecified policy classes, and LLMs assist with tasks such as modeling, coding, and decision analysis.

Can data, optimization, and LLMs instead be integrated into a systematic mechanism for designing effective replenishment policies? This question is important because inventory policy design involves two distinct choices. The first is the *policy class*: the functional form of the decision rule that maps the available inventory-state information to an ordering action. The second is the *parameter vector* that specifies a particular policy within that class. For example, the base-stock policy class is defined by a single-parameter decision rule, with each value of the base-stock level specifying a particular policy. Data characterize the operating environment, including demand patterns, lead times, service requirements, and cost tradeoffs. Given a policy class, optimization can then identify effective parameter values within that class. Neither data nor parameter optimization alone, however, determines which functional form should define the policy class. In practice, the policy class is often selected heuristically based on computational tractability or ease of interpretation, potentially at the expense of performance. This limitation motivates a broader question: *can LLMs expand the policy-design space beyond prespecified classes and support the automated discovery of previously unexplored policy classes, rather than selecting among or optimizing existing ones?*

The choice of policy class is particularly consequential in lost-sales inventory systems with positive lead times. In these systems, replenishment decisions must jointly account for on-hand inventory, outstanding pipeline orders, stochastic demand, and the irreversibility of lost sales. Although base-stock policies can perform well in some settings, they need not be optimal when lead times are positive because the value of an order depends on both the amount and timing of inventory already in the pipeline. Prior work has therefore examined alternative policy classes, including constant-order, capped base-stock, and projected-inventory-level policies (Zipkin 2008, Huh et al. 2009, Xin 2021, van Jaarsveld and Arts 2024, Liu et al. 2026, Xin 2026). This literature demonstrates that the choice of policy class can materially affect performance. It does not, however, provide a general mechanism for systematically searching beyond policy classes specified by researchers in advance.

LLMs create a new opportunity for policy-class search. Recent work shows that LLMs can generate executable code, interact with optimization models, and support automated heuristic design (Jiang et al. 2026, Chen et al. 2026, Romera-Paredes et al. 2024, Liu et al. 2024, Ye et al. 2024).

In particular, methods such as FunSearch, Evolution of Heuristics, ReEvo, and AlphaEvolve embed LLMs in evaluation-guided, reflective, or evolutionary search procedures that iteratively generate, evaluate, and refine executable algorithms (Romera-Paredes et al. 2024, Liu et al. 2024, Ye et al. 2024, Novikov et al. 2025). Concurrent work by Huang et al. (2026) applies LLM-based evolutionary search to generate interpretable inventory policies for online deployment under nonstationary demand, using reinforcement learning to evaluate and improve candidate policies. Baek (2026) studies a simple single-query protocol in which an LLM generates either an instance-specific solution or a reusable algorithm for well-specified problems in inventory control, queueing-network control, and assortment optimization, and finds that the strongest model is competitive with specialized methods. Baek et al. (2026) studies complementarities among operations-research heuristics, LLM agents, and human decision makers in multiperiod inventory control; in their setting, the LLM participates in or supports period-by-period ordering decisions rather than searching ex ante over stationary policy classes. Our setting and methodological focus differ from both streams. We study offline policy-class design for stationary lost-sales inventory systems and use off-the-shelf LLMs to generate parameterized policy classes, external numerical optimization to select parameter values within each generated class, and simulation to evaluate the resulting policies. This separation allows us to isolate whether optimization-guided feedback helps the LLM generate better policy classes, rather than merely better parameter values within previously generated classes.

Our Automated Inventory Policy Search (AIPS) framework builds on the discovery-oriented approach of recent LLM-based heuristic-design methods and adapts it to the sequential decision-making setting of inventory control. Given an inventory environment defined by demand data and instance-specific primitives, including lead time, holding cost, and lost-sales cost, AIPS uses an LLM to generate a pool of parameterized policy classes. Each class defines a mapping from the inventory state, including on-hand and pipeline inventory, to an order quantity. These classes may range from standard classes, such as the single-parameter base-stock class and the two-parameter (s, S) class, to richer classes with larger and more flexible parameter spaces. For each generated class, numerical optimization selects parameter values by minimizing a simulation-based empirical cost objective. The resulting optimized policy is then evaluated, and the best-performing policies, together with their class-defining code, optimized parameters, and performance summaries, are returned to the LLM to guide the generation of the next pool of candidate classes. Therefore, AIPS alternates between two interconnected steps: *policy-class search*, in which the LLM generates candidate policy classes, and *parameter optimization*, in which numerical optimization identifies effective parameter values within each generated class. Simulation-based evaluation closes the loop by measuring the performance of the optimized policies and feeding this information, together with the optimized parameter values, back to the LLM to guide subsequent policy-class search.

We conduct an extensive computational study of AIPS in lost-sales inventory systems with positive lead times. Rather than attempting to characterize optimal policies, we use a sequence of experiments to evaluate how effectively LLMs can generate useful classes of inventory policies and how the framework’s components contribute to performance. First, benchmark comparisons assess whether AIPS discovers policy classes containing policies that outperform optimized base-stock policies. Second, optimization ablations quantify the contribution of numerical parameter optimization and examine whether feeding optimized parameter values and policy performance back to the LLM improves subsequent policy-class search. Third, structural analyses identify the inventory-control motifs that recur in the best-performing discovered policies. Fourth, generalization experiments vary lead times, lost-sales-penalty-to-holding-cost ratios, and demand distributions to evaluate whether discovered policy classes remain effective beyond the instances on which they were developed. Fifth, cross-model comparisons measure the sensitivity of the framework’s performance to the choice of LLM backbone. Collectively, these experiments evaluate both the quality of the policies discovered by AIPS and the mechanisms through which optimization-guided feedback improves the search over policy classes.

1.1. Contributions and Related Literature

Our work contributes to several streams of research: lost-sales inventory control, data-driven inventory decision making, LLM-based automatic heuristic design, and approximate policy search. We organize our contributions around these connections.

- **Automated policy search for inventory control.** We develop a computational framework that expands inventory policy design beyond a prespecified menu of standard heuristics. The framework uses an off-the-shelf LLM to generate executable parameterized policy classes and numerical optimization to identify effective policies within those classes; the resulting optimized policies and performance information then guide subsequent structural search. Recent LLM-based automatic heuristic-design methods use evaluator-guided or evolutionary procedures to generate and improve executable algorithms (Romera-Paredes et al. 2024, Liu et al. 2024, Ye et al. 2024, Novikov et al. 2025). Concurrent inventory studies pursue related but distinct objectives: Huang et al. (2026) develop an end-to-end framework that uses a reinforcement-learning-trained LLM to evolve interpretable inventory policies with statistical safety guarantees for deployment, whereas Baek et al. (2026) study complementarities among LLM agents, operations-research heuristics, and human decision makers in sequential inventory decision making.

In contrast, we study the ex ante discovery of reusable policy classes for stationary lost-sales inventory systems. Our main contribution is to show how data and optimization can guide a general-purpose LLM to search a broad space of executable inventory decision rules and uncover

effective policy classes that need not be specified by the researcher in advance. More importantly, because both LLM and optimization components are modular and rely on off-the-shelf technologies, our framework can directly benefit from advances in LLM reasoning and code generation as well as improvements in the efficiency and capability of commercial optimization solvers, without requiring changes to its underlying architecture.

- **Exploratory computational study.** We provide systematic computational evidence on when LLM-guided inventory policy search creates value, why it does so, and how robust that value is across problem settings and LLM backbones. Our study varies key primitives of lost-sales inventory systems, including lead times from two to eight periods, multiple lost-sales-penalty-to-holding-cost ratios, and demand environments spanning bounded continuous, bounded discrete, light- to moderate-tailed unbounded, and heavy-tailed or highly skewed distributions. We also vary central features of the search procedure, most importantly whether LLM-generated policy classes undergo numerical parameter optimization, and compare performance across six LLM backbones: GPT-5 Nano, Gemini 2.5 Flash-Lite, Grok 4.1 Fast Non-Reasoning, GPT-5 Mini, DeepSeek V3, and Gemini 3 Flash. Existing studies of LLM-based heuristic design and LLM-enabled inventory decision making demonstrate that LLMs can generate executable decision rules and support sequential decisions, but offer limited evidence on their performance across heterogeneous inventory environments and algorithmic configurations. Our experiments address this gap by identifying when LLM-generated policy classes outperform optimized base-stock benchmarks, quantifying the contribution of numerical parameter optimization and optimization-guided feedback, characterizing recurring structural motifs in the discovered classes, evaluating their transfer across inventory environments, and measuring the sensitivity of the framework to the choice of LLM backbone.

- **Optimization-guided evaluation for LLM-based policy search.** We show that numerical optimization should be an integral part of the evaluation mechanism in LLM-guided policy search. Rather than evaluating the initial policy proposed by the LLM, we evaluate the best policy that can be obtained within each generated policy class by optimizing its parameters. Recent LLM-based heuristic-design methods use evaluators to execute candidate programs and guide reflective or evolutionary search (Romera-Paredes et al. 2024, Liu et al. 2024, Ye et al. 2024, Novikov et al. 2025). We extend this paradigm by introducing an optimization-guided evaluator that measures the quality of a policy class through its optimized representative rather than its initial parameterization. Computationally, this distinction proves important: optimization not only improves the performance of individual policies but also provides substantially more informative feedback, directing the LLM toward increasingly effective policy classes across generations.

- **Structurally meaningful discovered policies.** We show that AIPS discovers policy classes with interpretable and economically meaningful functional form rather than arbitrary black-box

decision rules. Across problem instances, the strongest classes repeatedly combine recognizable inventory-control motifs, including order caps, discounted or weighted representations of pipeline inventory, and threshold-based replenishment logic. These motifs relate naturally to established lost-sales policies: capped base-stock policies augment base-stock logic with order caps, while projected-inventory-level policies adjust replenishment decisions using anticipated future inventory positions (Xin 2021, van Jaarsveld and Arts 2024, Xin 2026). Our findings extend this literature by showing that such motifs need not be specified by the researcher *ex ante* but can emerge from LLM-guided search and be recombined into richer parameterized policy classes. More broadly, AIPS can be viewed as an evolutionary search over inventory-specific basis-function representations of replenishment policies: the LLM proposes transformations and combinations of inventory-state variables that define candidate policy classes, while numerical optimization selects their associated parameters. In this respect, our work contributes to recent research on automating approximation design for Markov decision processes (Pakiman et al. 2025, Nadarajah and Cire 2025, Adelman et al. 2025, Lagzi et al. 2026) by using LLMs to discover interpretable structural representations tailored to inventory control.

- **Strong and transferable discovered policy classes.** We show that AIPS can discover replenishment-policy classes containing policies that frequently outperform optimized base-stock benchmarks and remain effective beyond the instances on which they are generated. This comparison is consequential because base-stock policies are natural and widely used benchmarks, yet they can perform poorly in lost-sales systems with positive lead times. Prior work has therefore developed alternative policy classes, including constant-order, capped base-stock, and projected-inventory-level policies (Zipkin 2008, Huh et al. 2009, Xin 2021, van Jaarsveld and Arts 2024).

Rather than manually designing a new heuristic policy class for lost-sales inventory problems, we develop a mechanism that automatically generates policy classes with diverse structures, optimizes policies within those classes, and retains the strongest classes through simulation-based evaluation. We further assess whether the discovered classes capture reusable replenishment structures rather than instance-specific rules by transferring them across changes in lead time, cost ratio, and the demand distribution’s family, mean, and variability, with their parameters reoptimized for each new environment. This analysis goes beyond conventional out-of-sample evaluation on new demand realizations from a fixed environment and extends robustness questions in the lost-sales literature from prespecified policy classes to classes generated automatically through search. We also examine sensitivity to the LLM backbone and find that model choice affects the economic performance of the discovered policies, while numerical parameter optimization reduces, but does not eliminate, this variation. Collectively, these results indicate that the value of LLMs in inventory control lies not in making period-by-period ordering decisions directly, but in expanding the set of effective and transferable policy classes available for optimization and evaluation.

1.2. Organization

The remaining part of the paper is organized as follows. In §2, we introduce the lost-sales inventory model and the benchmark policies. In §3, we present the AIPS framework, including the LLM-based policy-generation step, the numerical optimization step, the simulation-based evaluation step, and the feedback mechanism across generations. In §4, we describe the experimental design, report the main benchmark and ablation results, and analyze the structural motifs that emerge in the discovered policies. In §5, we study transferability across lead times, cost ratios, and demand distributions. In §6, we examine the role of optimization. We use DeepSeek (specifically, `deepseek-chat-v3-0324`) as the default LLM backbone in §§4–6, holding the generator fixed to isolate the value of the proposed data–LLM–optimization framework relative to benchmark policies, its ability to uncover reusable structural motifs, and the role of the external optimizer. In §7, we relax this fixed-backbone design and evaluate the framework across multiple LLM backbones. We conclude in §8. Additional numerical results are reported in the online supplement. A GitHub repository will be made publicly available to facilitate potential future adaptations or extensions of AIPS.

2. Model and Optimization Objective

We consider a single-item, periodic-review inventory system over a finite horizon of T periods with deterministic lead time L . Time is indexed by $t = 1, \dots, T$. At the beginning of period t , before the order scheduled to arrive in that period is received, the inventory state is $\mathbf{I}_t := (I_t^{\text{on}}, I_{t,1}^{\text{pipe}}, \dots, I_{t,L}^{\text{pipe}})$. Throughout, we use boldface symbols to denote vectors. Here, I_t^{on} denotes the on-hand inventory before receipt of the arriving order, and $\mathbf{I}_t^{\text{pipe}} := (I_{t,1}^{\text{pipe}}, \dots, I_{t,L}^{\text{pipe}})$ denotes the pipeline-inventory vector. Its component $I_{t,\ell}^{\text{pipe}}$ is the quantity scheduled to arrive at the beginning of period $t + \ell - 1$; in particular, $I_{t,1}^{\text{pipe}}$ is the quantity arriving in period t . We assume that the initial inventory state $\mathbf{I}_1$ is known to the decision maker and that $I_1^{\text{on}} = 0$.

At the beginning of period t , the first pipeline component arrives, yielding available inventory $\bar{I}_t := I_t^{\text{on}} + I_{t,1}^{\text{pipe}}$. The decision maker then places an order $a_t \geq 0$, which enters the pipeline and arrives at the beginning of period $t + L$. An i.i.d. demand realization D_t is then drawn from distribution $\tilde{D}$, and any unmet demand is lost. Consequently, the quantity sold is $\min\{\bar{I}_t, D_t\}$, the next-period on-hand inventory is $I_{t+1}^{\text{on}} = (\bar{I}_t - D_t)^+$, and lost sales are $(D_t - \bar{I}_t)^+$, where $x^+ := \max\{x, 0\}$. At the end of the period, the remaining pipeline components shift forward by one position, and the newly placed order enters as the final component. The resulting state transition is

$$\mathbf{I}_{t+1} := \left((\bar{I}_t - D_t)^+, I_{t,2}^{\text{pipe}}, \dots, I_{t,L}^{\text{pipe}}, a_t \right), \quad \mathbf{I}_{t+1}^{\text{pipe}} := \left(I_{t,2}^{\text{pipe}}, \dots, I_{t,L}^{\text{pipe}}, a_t \right).$$

The period- t cost function is given by $c(\mathbf{I}_t, D_t) := h(\bar{I}_t - D_t)^+ + p(D_t - \bar{I}_t)^+$, where $h \geq 0$ and $p \geq 0$ are the unit holding and lost-sales costs, respectively. We normalize the variable ordering cost

to zero. Thus, the first term is the holding cost incurred on inventory remaining after demand is served, whereas the second term is the penalty incurred on unmet demand.

We focus on stationary replenishment policies represented by decision rules of the form $\pi : \mathcal{I} \rightarrow \mathcal{A}$, where $\mathcal{I} \subseteq \mathbb{R}_+^{L+1}$ is the inventory-state space and $\mathcal{A} := [0, \bar{a}]$ is the feasible action space, with $\bar{a} < \infty$ denoting an upper bound on the order quantity. In our simulations, we set $\bar{a}$ sufficiently large so that the upper bound does not bind. For each state $\mathbf{I} \in \mathcal{I}$, policy π prescribes an order quantity $\pi(\mathbf{I}) \in \mathcal{A}$. Stationarity means that the decision rule does not depend explicitly on the period index, while the Markov property means that past demands, states, and actions affect the current decision only through the current inventory state $\mathbf{I}$. Let Π denote the set of feasible stationary Markov replenishment policies. This set includes policies from standard classes, such as the single-parameter base-stock class and the two-parameter (s, S) class, as well as policies from richer parameterized classes with more flexible functional forms and larger parameter spaces. For a policy $\pi \in \Pi$, let $\mathbf{I}_t^\pi$ denote the inventory state in period t when policy π is followed. The objective is to solve the policy optimization problem

$$\inf_{\pi \in \Pi} J(\pi), \quad J(\pi) := \mathbb{E}_{\tilde{D}}^\pi \left[\sum_{t=1}^T c(\mathbf{I}_t^\pi, D_t) \right], \quad (\text{PO})$$

where the expectation is taken with respect to the demand distribution $\tilde{D}$ and policy π .

Tackling (PO) poses two fundamental challenges. First, the demand distribution $\tilde{D}$ is unknown, so the expectation in the objective function $J(\pi)$ cannot be evaluated directly. A standard approach is therefore to replace the expectation in (PO) with an empirical objective constructed from observed demand data, as studied in the data-driven inventory literature (e.g., [Ban and Rudin 2019](#), [Lin et al. 2022](#), [Zhang et al. 2025](#)). Second, the unrestricted policy space $\Pi := \{\pi : \mathcal{I} \rightarrow \mathcal{A}\}$ is an infinite-dimensional function space, making direct optimization over policies generally intractable. This difficulty is compounded by the curse of dimensionality: the inventory state includes both on-hand inventory and the entire pipeline, so its dimension grows with the lead time. Consequently, a broad literature restricts attention to tractable policy classes for lost-sales inventory systems and establishes conditions under which such policies perform near-optimally ([Zipkin 2008](#), [Huh et al. 2009](#), [Goldberg et al. 2016](#), [Xin 2021](#), [van Jaarsveld and Arts 2024](#)).

Data-driven cost function. For the first challenge, we approximate the expectation with respect to the unknown demand distribution $\tilde{D}$ using sample-average approximation. Let $\mathcal{D} := \{(\hat{D}_1^n, \dots, \hat{D}_T^n) : n = 1, \dots, N\}$ denote a dataset consisting of $N \geq 1$ demand sample paths, each of length T . We then define the empirical policy optimization problem as

$$\inf_{\pi \in \Pi} \hat{J}(\pi), \quad \hat{J}(\pi) := \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^T c(\mathbf{I}_t^{n,\pi}, \hat{D}_t^n), \quad (\text{EPO})$$

where $(\hat{D}_1^n, \dots, \hat{D}_T^n)$ is the n th observed demand trajectory and $\mathbf{I}_t^{n,\pi}$ is the inventory state at the beginning of period t when policy π is simulated along that trajectory. Thus, $\hat{J}(\pi)$ is the sample-average cost of policy π over the training trajectories and provides an empirical approximation of $J(\pi)$. Because all candidate policies are evaluated on the same demand trajectories, $\hat{J}$ provides a common empirical criterion for policy comparison.

Parametric policy class. For the second challenge, we approximate the unrestricted policy space Π , which contains all feasible mappings from inventory states to ordering decisions, by a parameterized policy class. Specifically, a *policy class* $\Pi_\Theta := \{\pi_\theta : \theta \in \Theta\} \subseteq \Pi$ is characterized by a *functional form* $\pi_\theta(\cdot) : \mathcal{I} \rightarrow \mathcal{A}$ and a *feasible parameter space* $\Theta \subseteq \mathbb{R}^d$. The functional form determines how the inventory state enters the ordering rule, while the parameter vector $\theta \in \Theta$ selects a particular policy within the class. Thus, for any fixed θ , $\pi_\theta(\cdot)$ is a fully specified policy that maps an inventory state $\mathbf{I} \in \mathcal{I}$ to the order quantity $\pi_\theta(\mathbf{I}) \in \mathcal{A}$. This representation gives rise to two distinct tasks: policy-class design, which determines the functional form and parameter space, and within-class parameter optimization, which selects parameter values for the inventory environment under consideration. For example, the base-stock policy class, denoted Π_Θ^{BS} , is given by the one-dimensional parameter $\theta := (\theta_B)$ and is defined as

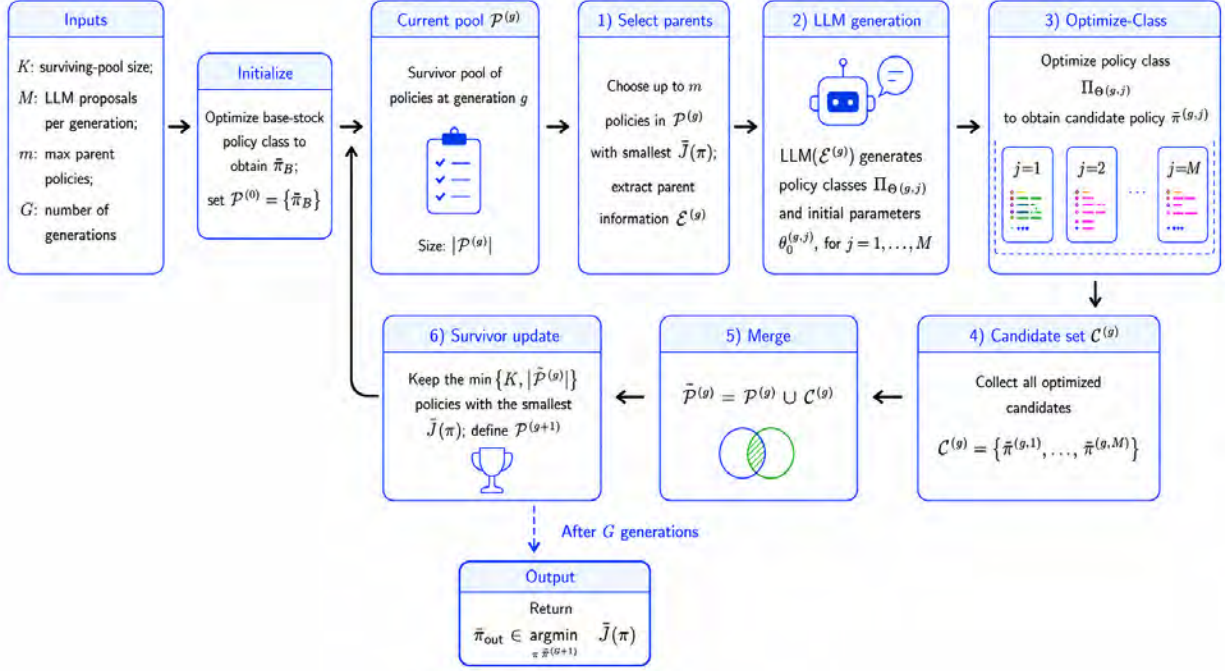
$$\Pi_\Theta^{\text{BS}} := \left\{ \pi_B : \pi_B(\mathbf{I}) = \min \left\{ \bar{a}, (\theta_B - I^{\text{pos}})^+ \right\}, \theta_B \in \Theta \right\},$$

where $I^{\text{pos}} := I^{\text{on}} + \sum_{\ell=1}^L I_\ell^{\text{pipe}}$ denotes the inventory position associated with state $\mathbf{I}$. Here, $\bar{a}$ continues to denote the upper bound on the order quantity. Each value $\theta_B \in \Theta$ therefore specifies one policy π_B within the base-stock class. The best policy within this class is obtained by selecting $\theta_B \in \Theta$ to minimize the population cost $J(\pi_B)$ when the demand distribution $\tilde{D}$ is known, or the empirical cost $\hat{J}(\pi_B)$ when only demand dataset $(\hat{D}_1^n, \dots, \hat{D}_T^n)$ is available.

3. LLM-Guided Inventory Policy Search

In this section, we present AIPS that integrates data, optimization, and LLMs. Specifically, given a finite search budget, at each search generation, an LLM proposes policy classes and an external numerical optimizer searches for low-cost parameter values within each class. These components are embedded in an evolutionary loop in which high-performing policies guide subsequent policy-class proposals. We present these components in the following order: §3.1 introduces policy-class design via the LLM, §3.2 introduces within-class parameter optimization, §3.3 combines the two components into the full evolutionary feedback loop, and §3.4 collects validation and numerical implementation details. Figure 1 provides an overview of the complete AIPS procedure.

Figure 1 Overview of the AIPS procedure. At each generation, AIPS selects parent policies from the current survivor pool, uses the LLM to generate new policy classes, optimizes the parameters of each generated class, and retains the best K policies for the next generation. After G generations, the best surviving policy is returned.



3.1. Policy-Class Design via an LLM

Rather than manually prespecifying a policy class (e.g., a base-stock policy), AIPS uses an LLM to iteratively generate parameterized policy classes. For each generated class, a numerical optimizer searches over its feasible parameter domain to identify a parameter vector that yields low empirical training cost, thereby producing an optimized policy within that class. High-performing optimized policies are then used to guide the generation of new policy classes. We refer to the policies selected for this purpose as *parent policies*; each parent is therefore an instantiated policy from a previously generated class, together with its optimized parameter vector. In §3.3, we specify how these parent policies are selected. Given one or more parents, the prompt provides the LLM with each parent’s functional form, optimized parameter vector, and empirical training cost. Using this information, the LLM is asked to generate exactly one new parameterized policy class, together with an initial parameter vector and a feasible parameter domain.

Illustrative LLM prompt. To make the LLM interface concrete, consider a policy-class generation call in which a base-stock policy and a constant-order policy serve as parent policies. Figure 2 presents an abridged version of the prompt supplied to the LLM. The prompt contains four top-level blocks. The **PROBLEM INSTANCE STATEMENT** specifies the inventory instance introduced in §2, including the lead time L , the unit holding and lost-sales costs h and p , and the demand dataset $\mathcal{D}$. The **TASK DESCRIPTION** defines the generation objective: using the parent policies as

Figure 2 Illustrative LLM prompt for generating a new parameterized policy class from parent policies.

```

PROBLEM INSTANCE STATEMENT
Lead time: L = 6
Holding cost: h = 1
Lost-sales cost: p = 2
Demand data: demand sample paths

TASK DESCRIPTION
Given parent policies, generate a new policy class that achieves lower empirical training cost on
the demand data.
The functional form of the policy class should have the following input and output format:
Input: current on-hand inventory and pipeline inventory
Output: one nonnegative scalar order quantity

PARENT POLICY SET
PARENT POLICY 1
Functional form of the base-stock policy:
...
Parameter vector for base-stock policy:
...
Empirical training cost:
...
PARENT POLICY 2
Functional form of the constant order policy:
...
Parameter vector for constant order policy:
...
Empirical training cost:
...

REQUIREMENT
Return exactly one new candidate policy class consisting of:
1. functional form of the policy class by Python code;
2. an initial value for every tunable parameter; and
3. a feasible lower and upper bound for every parameter.

```

guidance, the LLM is asked to propose a new parameterized policy class that, after parameter optimization, can yield a policy with lower empirical cost on the given demand data. The **PARENT POLICY SET** provides the feedback used to guide the structural search. For each parent policy $\hat{\pi}$, the prompt reports its functional form, optimized parameter vector, and corresponding empirical cost. Finally, the **REQUIREMENT** block specifies the output of a single LLM proposal call. The LLM returns (Π_{Θ}, θ_0) , where $\Pi_{\Theta} = \{\pi_{\theta} : \theta \in \Theta\}$ is the generated policy class, Θ is its feasible parameter domain, and $\theta_0 \in \Theta$ is the LLM-supplied initial parameter vector. The example omits formatting and validation instructions that do not affect this conceptual input–output relationship. The complete prompt template is reported in [EC.1](#).

The LLM prompt in Figure 2 can be formalized by representing the information supplied about the parent policies. Let

$$\mathcal{E} := \left\{ \left(\Pi_{\Theta_j}, \hat{\theta}_j, \hat{\mathcal{J}}(\pi_{\hat{\theta}_j}) \right) \right\}_{j=1}^m$$

denote the parent set. Each element records a previously generated policy class Π_{Θ_j} , the optimized parameter vector $\hat{\theta}_j$ that instantiates a parent policy $\pi_{\hat{\theta}_j}$ within that class, and the resulting empirical training cost. Thus, $\mathcal{E}$ contains both the structural information and the performance feedback shown to the LLM. Given this parent set, one LLM call produces a new parameterized policy class and an initial parameter vector:

$$(\Pi_{\Theta}, \theta_0) \leftarrow \text{LLM}(\mathcal{E}),$$

where Π_{Θ} is the newly proposed policy class and $\theta_0 \in \Theta$ is the LLM-supplied initialization for subsequent parameter optimization. The inventory instance, training data, and prompt instructions are fixed within a search run and are therefore suppressed from the arguments of LLM. Thus, the LLM primarily searches over policy structures rather than parameter values within a fixed structure. It may modify the state transformations, algebraic operations, branching logic, number of tunable parameters, and their roles. The LLM-supplied θ_0 serves only as an initialization; proposal quality is evaluated after the external optimizer searches for low-cost parameters within Π_{Θ} .

3.2. Within-Class Parameter Optimization

Given an LLM output (Π_{Θ}, θ_0) , the optimization module searches over Θ while holding the functional form $\pi_{\theta}(\cdot)$ fixed. The corresponding within-class empirical optimization problem is

$$\min_{\theta \in \Theta} \hat{J}(\pi_{\theta}) = \min_{\theta \in \Theta} \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^T c\left(\mathbf{I}_t^{n, \pi_{\theta}}, \hat{D}_t^n\right). \quad (\text{EPO}(\Pi_{\Theta}))$$

Problem $(\text{EPO}(\Pi_{\Theta}))$ is the empirical policy-optimization problem (EPO) restricted to the generated class Π_{Θ} . Every parameter vector is evaluated on the same fixed training trajectories, so differences in empirical cost are not caused by demand resampling.

Because the numerical search is conducted under a finite computational budget, we do not assume that it solves problem $(\text{EPO}(\Pi_{\Theta}))$ to local or global optimality. Instead, among the parameter vectors successfully evaluated by the numerical routine, let $\hat{\theta}$ denote one attaining the lowest empirical cost. We define

$$\hat{\pi} := \pi_{\hat{\theta}}$$

as the retained policy associated with Π_{Θ} . We summarize the numerical procedure by the function

$$\hat{\pi} \leftarrow \text{Optimize}(\Pi_{\Theta}, \theta_0). \quad (1)$$

The external optimizer may change only θ ; it cannot alter the policy’s state variables, transformations, algebraic operations, branch conditions, or any other structural feature encoded by $\pi_{\theta}(\cdot)$. Consequently, LLM searches across policy classes, whereas **Optimize** searches for a policy within a fixed class.

Algorithm 1: Automated Inventory Policy Search (AIPS)

Receive: Maximum surviving-pool size K ; number of LLM proposals per generation M ; maximum number of parent policies m ; and number of generations G .

Initialize: Optimize the base-stock policy class to obtain $\hat{\pi}_B$, and set $\mathcal{P}^{(0)} \leftarrow \{\hat{\pi}_B\}$.

```

1 for  $g = 0, 1, \dots, G - 1$  do
2    $\mathcal{C}^{(g)} \leftarrow \emptyset$ 
3    $\mathcal{E}^{(g)} \leftarrow$  the parent information of the  $\min\{m, |\mathcal{P}^{(g)}|\}$  policies  $\pi \in \mathcal{P}^{(g)}$  with the smallest  $\hat{J}(\pi)$ 
4   for  $j = 1, \dots, M$  do
5     Generate  $(\Pi_{\Theta^{(g,j)}}, \theta_0^{(g,j)}) \leftarrow \text{LLM}(\mathcal{E}^{(g)})$  and compute  $\hat{\pi}^{(g,j)} \leftarrow \text{Optimize}(\Pi_{\Theta^{(g,j)}}, \theta_0^{(g,j)})$ 
6     Set  $\mathcal{C}^{(g)} \leftarrow \mathcal{C}^{(g)} \cup \{\hat{\pi}^{(g,j)}\}$ 
7   Define  $\tilde{\mathcal{P}}^{(g)} \leftarrow \mathcal{P}^{(g)} \cup \mathcal{C}^{(g)}$ 
8   Update  $\mathcal{P}^{(g+1)}$  to the  $\min\{K, |\tilde{\mathcal{P}}^{(g)}|\}$  policies  $\pi \in \tilde{\mathcal{P}}^{(g)}$  with the smallest values of  $\hat{J}(\pi)$ 

Return:  $\hat{\pi}_{\text{out}} \in \arg \min_{\pi \in \mathcal{P}^{(G)}} \hat{J}(\pi)$ .

```

3.3. Algorithm

Algorithm 1 combines the two components introduced above—LLM-based policy-class generation and numerical parameter optimization—within an evolutionary search. The algorithm maintains a surviving pool $\mathcal{P}^{(g)}$ of optimized policies at generation g . Starting from an optimized base-stock policy, each generation proceeds sequentially as follows. First, the best-performing policies in the current surviving pool are selected as parents (line 3). Second, for each of M offspring, the LLM uses the parent information to propose a new parameterized policy class, and the external optimizer searches within that class to obtain an optimized policy; the resulting policy is then added to the offspring set (line 6). Finally, the offspring and incumbent policies are pooled, and the K policies with the lowest empirical training costs survive to the next generation (lines 7–8). After G generations, the algorithm returns the best policy remaining in the surviving pool. The search is governed by four hyperparameters: K , the maximum size of the surviving pool; M , the number of LLM proposals generated in each generation; m , the maximum number of parent policies supplied to each LLM call; and G , the number of generations.

Initialization. The algorithm begins from a standard policy class that provides a valid incumbent before the first LLM call. Throughout the paper, we use the base-stock policy as this initializer. For target level θ_B , its ordering rule is $\pi_B(\mathbf{I}_t; \theta_B) := \left[\theta_B - \left(I_t^{\text{on}} + \sum_{k=1}^L I_{t,k}^{\text{pipe}} \right) \right]^+$. We treat this rule as a single-parameter policy class and apply the same numerical optimization module used for LLM-generated classes to determine its parameter. Let $\hat{\pi}_B$ denote the resulting optimized base-stock policy. The initial surviving pool is therefore $\mathcal{P}^{(0)} := \{\hat{\pi}_B\}$. This initialization ensures that the search starts from a feasible and interpretable policy and that the surviving pool is nonempty when the

first generation begins. Other user-specified policy classes could be used as initializers, although doing so may change the subsequent search trajectory.

Parent selection. At the beginning of generation g , the algorithm creates an empty offspring set $\mathcal{C}^{(g)}$ and selects parent policies from the current surviving pool $\mathcal{P}^{(g)}$. Specifically, it selects the $\min\{m, |\mathcal{P}^{(g)}|\}$ policies with the smallest empirical training costs. As described in the preceding subsection, the information supplied to the LLM for each selected parent consists of its generating policy class, optimized parameter vector, and empirical training cost. We collect this information in the parent set $\mathcal{E}^{(g)}$. Thus, the LLM observes both the structures that have performed well so far and the parameter values and costs obtained after numerical optimization.

Policy-class generation and parameter optimization. Holding the parent set $\mathcal{E}^{(g)}$ fixed, the algorithm makes M separate LLM calls. On proposal j , the LLM generates

$$\left(\Pi_{\Theta^{(g,j)}}, \theta_0^{(g,j)}\right) \leftarrow \text{LLM}(\mathcal{E}^{(g)}),$$

where $\Pi_{\Theta^{(g,j)}}$ is a new parameterized policy class and $\theta_0^{(g,j)} \in \Theta^{(g,j)}$ is its LLM-supplied initialization. The external optimizer then searches over the feasible parameter domain $\Theta^{(g,j)}$ to obtain an optimized policy

$$\hat{\pi}^{(g,j)} \leftarrow \text{Optimize}\left(\Pi_{\Theta^{(g,j)}}, \theta_0^{(g,j)}\right).$$

Importantly, the object retained by the algorithm is this optimized policy, not the policy obtained directly from the LLM-supplied initialization. Each optimized policy is added to the offspring set, giving $\mathcal{C}^{(g)} := \{\hat{\pi}^{(g,j)} : j = 1, \dots, M\}$.

Survival selection and population update. After all M proposals have been optimized, the offspring are combined with the policies that survived from the previous generation: $\tilde{\mathcal{P}}^{(g)} := \mathcal{P}^{(g)} \cup \mathcal{C}^{(g)}$. The policies in $\tilde{\mathcal{P}}^{(g)}$ are ranked according to their empirical training costs, and the next surviving pool $\mathcal{P}^{(g+1)}$ consists of the $\min\{K, |\tilde{\mathcal{P}}^{(g)}|\}$ lowest-cost policies. Hence, a newly generated policy survives only if its optimized performance is sufficiently competitive with that of the incumbents. Because the surviving policies become the candidates for parent selection in the next generation, better-performing policy structures receive greater opportunity to influence subsequent LLM proposals.

Output. After G generations, the algorithm returns the policy with the lowest empirical training cost among the final survivors: $\hat{\pi}_{\text{out}} \in \arg \min_{\pi \in \mathcal{P}^{(G)}} \hat{J}(\pi)$. Thus, selection throughout the search is based on performance *after* within-class parameter optimization. The external optimizer therefore plays two roles: it calibrates the parameters of each proposed class, and, through the resulting empirical costs, it determines which policy structures are retained and subsequently fed back to the LLM. This separation allows the LLM to focus primarily on searching over policy structures while the numerical optimizer searches for effective parameter values within each proposed structure.

3.4. Validation and Numerical Implementation Details

To operationalize AIPS, the implementation must also determine whether a generated policy class is admissible and whether a particular parameter vector can be executed reliably in simulation. We therefore apply validation at two levels—class-level static validation and parameter-specific runtime validation—and then summarize the numerical routine used for within-class search. These implementation safeguards are important for executability and reproducibility, but do not alter the conceptual search procedure described above.

Class-level static validation. Before numerical search, the implementation parses the generated source code and the annotations identifying optimizable parameters. A proposal is discarded if the code cannot be parsed; if the parameter metadata are missing or internally inconsistent; if $\theta_0 \notin \Theta$; or if the declared parameter domain is invalid. The policy function must satisfy the common interface: it may use only the current on-hand and pipeline-inventory state, must return a scalar order quantity, and may not depend explicitly on the period index, trajectory index, previous function calls, or persistent hidden state. Determinism and stationarity are enforced through hard-coded validation rules. A proposal that fails any of these checks causes `Optimize` to return $\emptyset$ without beginning numerical search.

Parameter-specific runtime validation. Passing class-level static validation does not imply that every $\theta \in \Theta$ produces an executable policy. Each parameter vector queried by the numerical routine is therefore checked during simulation. A parameter vector is considered feasible only if every training trajectory completes within the prespecified evaluation time limit and every visited state produces a finite scalar order quantity. Execution errors, timeouts, and nonfinite outputs make that parameter vector invalid. If $\mathcal{V} = \emptyset$, `Optimize` returns $\emptyset$, and the proposal does not enter the offspring pool.

Numerical implementation. In the implementation, we use the `minimize` routine in `scipy.optimize` with L-BFGS-B, finite-difference gradient estimates, bounded parameter domains, and 15 optimizer iterations. The simulation objective can be nonsmooth because of branch conditions, lost-sales dynamics, clipping, type conversions, and integer rounding. We therefore use L-BFGS-B as a standardized bounded numerical search routine, not as a method guaranteed to identify a local or global optimum. Among all valid parameter vectors evaluated under the budget, the routine retains the one with the lowest empirical training cost, as defined in (EPO).

4. Performance and Structure of Discovered Policies

This section evaluates the performance and structure of the policies discovered by AIPS. In §4.1, we define the 30 benchmark instances, describe the experimental design, and introduce the performance metrics used to compare AIPS with the optimized base-stock policy. In §4.2, we quantify the cost

reductions achieved by AIPS over generations and examine how these gains vary with lead time, cost parameters, and demand distributions. In §4.3, we characterize recurring structural motifs in the discovered policies by AIPS. In §4.4, we trace a representative evolutionary run of AIPS to illustrate how the incumbent policy evolves across generations and how structural motifs are combined and refined throughout the search.

4.1. Problem Instances and Performance Metrics

We consider 30 instances of the lost-sales inventory control problem in §2 to benchmark AIPS. Each instance corresponds to a unique combination of demand distribution, lead time, and cost parameters. Specifically, we consider three lead times $L \in \{2, 4, 6\}$, two cost settings $(h, p) \in \{(1, 2), (1, 5)\}$, and five demand specifications: Poisson demand, Exponential demand, and three Normal demand settings with standard deviations 10, 30, and 50. The mean demand is normalized to 100 in all cases. Thus, the benchmark set spans a range of operating regimes, from short- to long-lead-time systems and from relatively stable to highly variable demand environments.

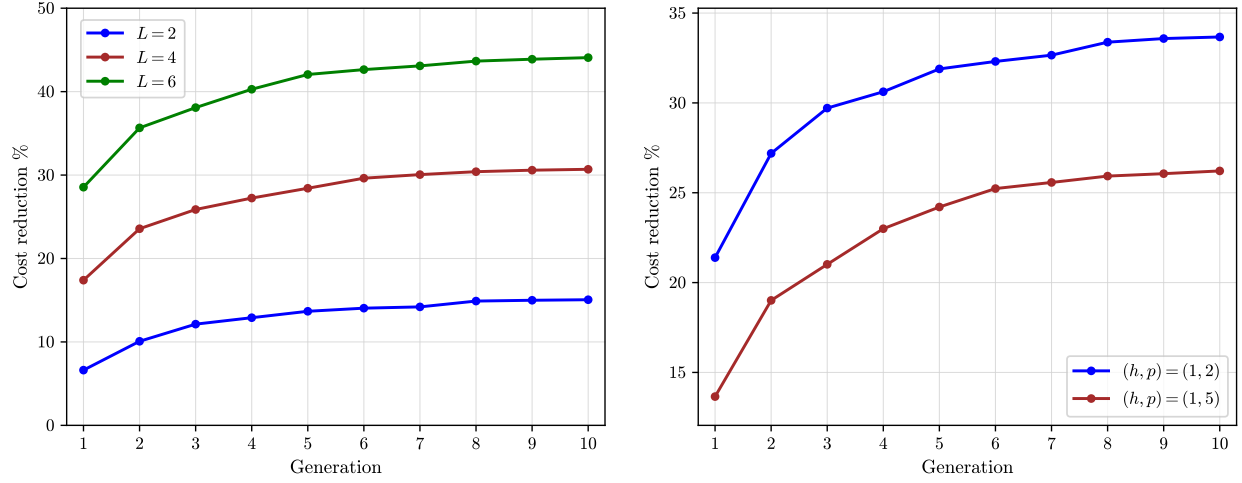
For each instance k , we run the evolutionary search for $G = 10$ generations. In each generation, the LLM generate $M = 10$ offsprings. We refer to one complete run of the algorithm over all G generations as a *repeat*. For each instance, we conduct 10 independent repeats using the same training data. In other words, the repeats differ only in the randomness of LLM sampling. For repeat r , let $\mathcal{P}_{k,r}^{(g)}$ denote the surviving pool after generation g . We rank the policies in this pool by empirical training cost and define the lowest empirical training cost attained through generation g in repeat r as

$$\hat{J}_{k,r}^{(g)} := \min_{\pi \in \mathcal{P}_{k,r}^{(g)}} \hat{J}_k(\pi).$$

For each instance k , we compare the policies discovered by AIPS with the optimized base-stock policy for the same instance. For each instance k , let $\hat{J}_k(\cdot)$ denote the empirical training cost evaluated on the same training trajectories used by AIPS. Using the same notation in Algorithm 1, the empirical cost of the optimized base-stock policy is $\hat{J}_k(\hat{\pi}_B)$, which will be used as the benchmark. We define the percentage cost reduction for instance k at generation g in repeat r as

$$\text{CR}_{k,r}^{(g)} := 100 \left(1 - \frac{\hat{J}_{k,r}^{(g)}}{\hat{J}_k(\hat{\pi}_B)} \right).$$

Thus, $\text{CR}_{k,r}^{(g)} > 0$ indicates that the best policy discovered after g generations has a lower cost than the optimized base-stock cost for instance k , with larger values corresponding to greater cost reductions. In this section, we often report the mean and standard error of $\text{CR}_{k,r}^{(g)}$ across all ten repeats and a family of problem instances with certain features (e.g., all instances with Poisson demand).

Figure 3 Impact of lead time (left) and cost parameters (right) on the cost reductions achieved by AIPS.

4.2. Performance Results

We first report some summary statistics of all the generated policies (see Table EC.1 in Appendix EC.2 for a detailed reference). Most notably, the performance improves steadily with generation: when averaging across all instances and repeats, CR increases from 17.5% in generation 1 to 30.0% in generation 10, while the median increases from 11.2% to 26.0%. Moreover, all reductions are strictly positive throughout the search, and by generation 10 they range from 5.2% to 72.7%. These results provide two main insights. First, AIPS consistently discovers policies that outperform the numerically optimized base-stock benchmark across a heterogeneous set of inventory instances, indicating that the gains are not driven by only a few favorable settings. Second, most of the improvement occurs early in the evolutionary search: 84.8% of the total increase in the generation-level mean from generations 1 to 10 is realized by generation 5 and 90.6% by generation 6. Thus, the search rapidly identifies effective policy structures in the first few generations and delivers smaller, incremental refinements.

Next, we examine how the gains from AIPS vary across problem characteristics. Figure 3 reports the generation-level mean of cost reduction, with generation on the horizontal axis and cost reduction percentage on the vertical axis. The left panel groups the instances based on lead time: the blue line corresponds to all instances with $L = 2$, the brown line denotes $L = 4$, and the green line denotes $L = 6$. Cost reduction increases with generation for all three lead times, but the magnitude differs substantially: by generation 10, the mean reduction is 15.1% for $L = 2$, 30.7% for $L = 4$, and 44.1% for $L = 6$. The right panel performs the same comparison across cost settings, with the blue line corresponding to all instances with $(h, p) = (1, 2)$ and the brown line denotes $(h, p) = (1, 5)$. By generation 10, the corresponding mean reductions are 33.6% and 26.3%, respectively. These patterns indicate that the value of richer policy structures increases markedly with lead time and, to a lesser extent, when holding inventory is relatively more costly than incurring lost sales. One

interpretation is that, as lead time grows, the base-stock policy’s aggregation of the pipeline into a single inventory-position measure becomes increasingly restrictive, creating greater scope for policies that respond more selectively to the composition of the pipeline. Similarly, the larger gains under $(h, p) = (1, 2)$ suggest greater value from refining inventory adjustments when excess inventory is relatively costly. We return to these structural mechanisms in §5.

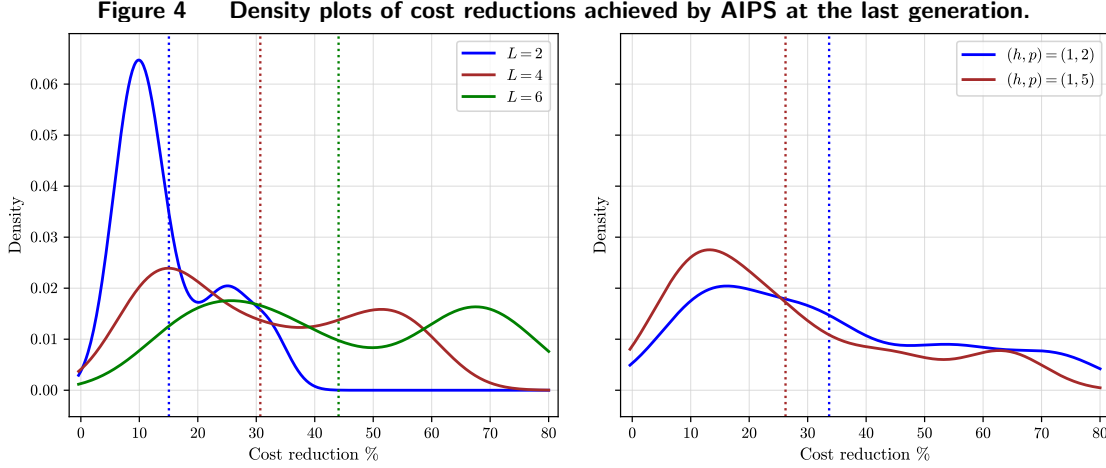


Figure 4 complements the generation-level averages in Figure 3 by showing the fitted density curve and the average value (represented by the dotted lines) of repeat-level cost reductions at the end of the evolutionary search. The left panel echoes the previous finding with a rightward shift as lead time increases: the mean cost reduction rises from (15.1%) for $(L = 2)$ to (30.7%) for $(L = 4)$ and (44.1%) for $(L = 6)$. The broader and partly multimodal densities for $(L = 4)$ and $(L = 6)$ indicate that the magnitude of the improvement remains heterogeneous across demand environments and repeats; nevertheless, the shift in the overall distributions, rather than only in their means, shows that the larger gains under longer lead times are not driven by a small number of extreme outcomes. This pattern is consistent with the structural interpretation developed in the paper: as the pipeline lengthens, the base-stock policy’s aggregation of all outstanding orders into a single inventory-position measure becomes increasingly restrictive, creating greater value for policies that distinguish among pipeline positions and moderate their replenishment responses. The right panel shows a similar, although less pronounced, effect of the cost parameters: the distribution under $((h, p) = (1, 2))$ is shifted to the right of that under $((h, p) = (1, 5))$, with respective means of (33.7%) and (26.2%). Their substantial overlap indicates that demand characteristics and lead time remain important sources of heterogeneity within each cost setting, but the overall shift supports the interpretation that richer policy structures are particularly valuable when holding excess inventory is relatively costly compared with incurring lost sales.

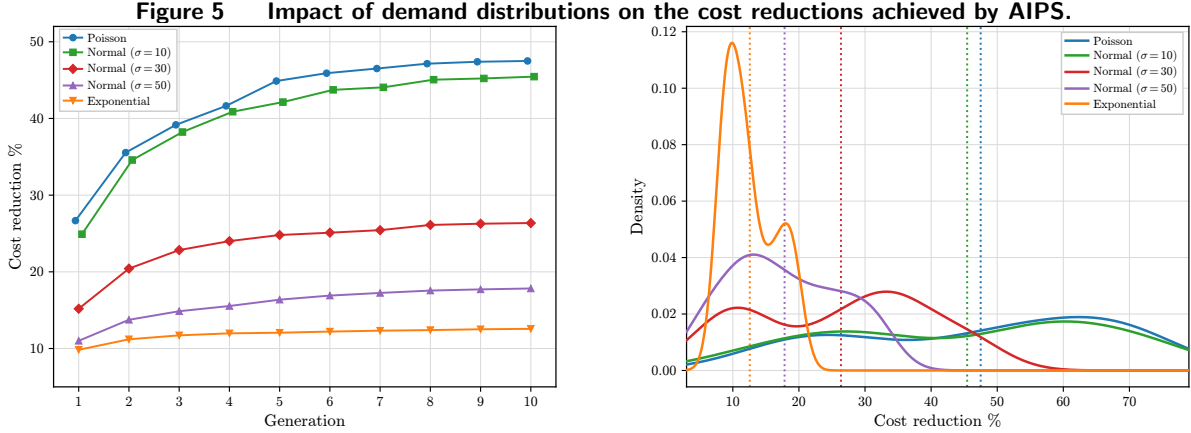


Figure 5 examines heterogeneity across demand specifications. In both panels, the five curves correspond to Poisson demand, Exponential demand, and Normal demand with standard deviations $\sigma \in \{10, 30, 50\}$. In the left panel, mean cost reduction remains positive and increases with generation under all five specifications, but its magnitude varies substantially. By generation 10, the mean reduction reaches approximately 48% under Poisson demand and 45% under Normal demand with $\sigma = 10$, compared with 27%, 18%, and 12% under Normal demand with $\sigma = 30$, Normal demand with $\sigma = 50$, and Exponential demand, respectively. Within the Normal family, the gains decline monotonically with demand variability, suggesting that the richer policy structures discovered by AIPS are particularly valuable when demand is relatively stable. The similar performance under Poisson demand and Normal demand with $\sigma = 10$, which have comparable means and standard deviations, further suggests that the discrete-versus-continuous distinction is less important in these low-variability environments. In contrast, the comparatively small gain under Exponential demand is consistent with the greater difficulty of exploiting pipeline information when demand is highly variable and skewed. The right panel shows the fitted densities of $CR_{k,r}^{(10)}$, with the dotted lines marking the corresponding means. The densities exhibit the same ordering as the generation-level averages. Thus, the differences in average performance reflect broad shifts in the distributions rather than a small number of unusually successful repeats. The broad and partly multimodal densities also reveal heterogeneity across lead times and cost settings within each demand model.

4.3. Structural Analysis of Discovered Policies

The previous results show that AIPS can produce policies with substantially lower empirical cost than the optimized base-stock policy across different settings. We now ask what these policies look like. This is important because a discovered policy is most useful when its functional form can be interpreted and related to known replenishment logic. In particular, we aim to answer three questions: First, what recurring design motifs appear in the generated policies? Second, which motifs are most common among top-ranked and later-generation policies? Third, do the discovered

policies look like opaque black-box code, or do they recombine interpretable building blocks of inventory policy?

We begin by defining the structural motifs that appear in the discovered policies. We say that a policy contains a given motif if its functional form includes the corresponding structural feature, allowing for algebraically equivalent expressions and differences in variable names across the generated Python code.

- (a) **Inventory position motif.** The policy depends on the total inventory position $I^{\text{on}} + \sum_{k=1}^L I_{t,k}^{\text{pipe}}$. This is the core state statistic used by the classical base-stock policy.
- (b) **Pipeline weighting motif.** The policy uses a *linearly weighted* pipeline term $\sum_{k=1}^L w_k I_{t,k}^{\text{pipe}}$. This motif modifies how outstanding orders enter the inventory signal. The base-stock policy gives each pipeline unit the same credit. This aggregation is potentially restrictive, because orders arriving sooner and orders arriving later need not have the same value for avoiding lost sales over the next few periods. Weighted pipeline terms allow the policy to assign different credits to different parts of the pipeline.
- (c) **Nonlinear pipeline composition motif.** The policy computes an inventory signal using a *nonlinear* function $\varphi(\mathbf{I}_t^{\text{pipe}})$ of the pipeline vector. Examples include $\varphi(\mathbf{I}_t^{\text{pipe}}) = \max_k I_{t,k}^{\text{pipe}}$, $\varphi(\mathbf{I}_t^{\text{pipe}}) = \min_k I_{t,k}^{\text{pipe}}$, and $\varphi(\mathbf{I}_t^{\text{pipe}}) = (\sum_{k=1}^m I_{t,k}^{\text{pipe}}) / (\sum_{k=1}^L I_{t,k}^{\text{pipe}})$ for some $m < L$. This motif further generalizes the pipeline weighting motif by introducing non-linearity into the order decision.
- (d) **Order-up-to motif.** The order quantity is the *exact non-negative gap* between a fixed target level η and a *state-dependent inventory signal* $f(\mathbf{I}_t)$. That is, $(\eta - f(\mathbf{I}_t))^+$. This motif (together with the inventory position motif) subsumes the base-stock rule. It introduces a target-seeking negative-feedback structure: when the inventory signal falls below the target, the policy orders enough to close the gap.
- (e) **Partial adjustment motif.** The order quantity is a *fraction of* the non-negative gap between a fixed target level and an inventory signal: $\alpha (\eta - f(\mathbf{I}_t))^+$, $\alpha \in (0, 1)$. Relative to the order-up-to motif, partial adjustment changes the aggressiveness of the replenishment response. Under base-stock policy, every positive gap is closed immediately. With positive lead times, such one-step correction can be too aggressive: a large order placed today arrives only after a delay, while additional orders may be placed before the effect of the first order is observed. Partial adjustment closes the gap gradually and can reduce overshooting.
- (f) **State-dependent order-up-to motif.** This motif generalizes the order-up-to motif by allowing the target to depend on the state: $(\eta(\mathbf{I}_t) - f(\mathbf{I}_t))^+$. The target may change with on-hand inventory or with features of the pipeline, rather than remaining fixed across all states.

-
- (g) **State-dependent partial adjustment motif.** This motif combines a state-dependent target with partial adjustment: $\alpha(\eta(\mathbf{I}_t) - f(\mathbf{I}_t))^+$. It allows the policy to change both the target and the strength of the response to current states.
 - (h) **Constant order motif.** The policy contains a state-independent baseline replenishment component b_0 . This component may appear as an additive term, as a lower bound, or as a constant amount ordered in a subset of states. A positive baseline keeps inventory flowing into the pipeline and can prevent the system from waiting too long before restarting replenishment.
 - (i) **Order clipping motif.** The policy imposes a fixed lower or upper bound on the order quantity, for example, $\min\{\bar{\theta}, \max\{\underline{\theta}, g(\mathbf{I}_t)\}\}$. Clipping regularizes extreme order quantities. An upper clip prevents very large corrective orders in depleted states, while a lower clip prevents replenishment from becoming too small in states where a continued inventory flow is desirable.
 - (j) **Order smoothing motif.** The policy explicitly uses the previous order quantity, which is represented by the last component $q_{t,L}$ of the current pipeline vector after initialization. Examples include $\min\{g(\mathbf{I}_t), I_{t,L}^{\text{pipe}} + \Delta\}$, $\max\{g(\mathbf{I}_t), I_{t,L}^{\text{pipe}} - \Delta\}$, and $\beta g(\mathbf{I}_t) + (1 - \beta)I_{t,L}^{\text{pipe}}$. This motif avoids abrupt changes in order quantities across adjacent periods while preserving stationarity, because $I_{t,L}^{\text{pipe}}$ is part of the current state.
 - (k) **Thresholding motif.** The policy reacts only when a state-dependent signal crosses a threshold, for example, $f(\mathbf{I}_t) \cdot \mathbf{1}\{g(\mathbf{I}_t) > \tau\}$. Even without a fixed ordering cost, thresholding can be useful in a lost-sales system with lead times. Small inventory gaps may reflect temporary noise, and reacting to all of them immediately can generate replenishment that arrives after the imbalance has already disappeared.

Policies generated by AIPS may differ in algebraic form, parameter names, or code implementation while still sharing the same underlying replenishment logic. We therefore interpret the motifs above as recurring structural building blocks rather than literal code patterns. For example, the LLM may introduce parameters named `smoothing_factor`, `aggressiveness`, or `order_scaler`, even though each implements the same partial-adjustment motif. These motifs are not mutually exclusive; most discovered policies combine several of them within a single replenishment rule.

Viewed through the lens of the base-stock policy, these motifs primarily enrich two aspects of the decision rule. The first is *state representation*. The base-stock policy summarizes the pipeline through the aggregate quantity $\sum_{k=1}^L I_{t,k}^{\text{pipe}}$, thereby assigning the same weight to all outstanding orders regardless of when they arrive. The discovered policies can instead use weighted, nonlinear, or otherwise state-dependent summaries of the pipeline, allowing the ordering decision to distinguish among different pipeline configurations. The second is *inventory adjustment*. A base-stock policy responds to a positive target gap by attempting to close it immediately. The discovered policies can

modify this response through partial adjustment, state-dependent targets, clipping, thresholding, smoothing, or baseline replenishment. Thus, the search can change both *how the inventory state is summarized* and *how the resulting signal is translated into an order quantity*.

More broadly, these motifs can be viewed as structural *basis functions* for constructing policy approximations, analogous to the basis elements used in approximate dynamic programming to obtain tractable approximations of policies or value functions (Powell 2011). From this perspective, AIPS automates an important part of policy approximation: rather than requiring a domain expert to prespecify the relevant structural bases and how they should be combined for a given problem instance, the LLM searches over these building blocks and their combinations, while the numerical optimizer calibrates the associated parameters.

Table 1 Prevalence of structural motifs in manually labeled generated policies.

Motif	All ($n = 100$)	Top-3 Ranked ($n = 40$)	Generations 8–10 ($n = 23$)
Inventory position	77 (77.0%)	30 (75.0%)	18 (78.3%)
Pipeline weighting	49 (49.0%)	19 (47.5%)	12 (52.2%)
Nonlinear pipeline composition	28 (28.0%)	14 (35.0%)	9 (39.1%)
Order-up-to	30 (30.0%)	11 (27.5%)	6 (26.1%)
Partial adjustment	27 (27.0%)	8 (20.0%)	6 (26.1%)
State-dependent order-up-to	17 (17.0%)	8 (20.0%)	4 (17.4%)
State-dependent partial adjustment	33 (33.0%)	15 (37.5%)	11 (47.8%)
Constant order	13 (13.0%)	5 (12.5%)	5 (21.7%)
Order clipping	37 (37.0%)	15 (37.5%)	13 (56.5%)
Order smoothing	7 (7.0%)	4 (10.0%)	3 (13.0%)
Thresholding	13 (13.0%)	3 (7.5%)	2 (8.7%)

Notes: Entries report the number of manually labeled policies containing each motif, with the corresponding percentage in parentheses. Because a policy may contain multiple motifs, the percentages need not sum to 100.

Table 1 summarizes a manual coding exercise designed to identify which structural motifs recur in the policies generated by AIPS. We first randomly sample 100 generated policies. For each sampled policy, we manually inspect its Python code to understand the proposed functional form, simplify this replenishment logic when necessary, and record a binary indicator for whether each motif defined above is present. A motif is counted only if it remains active after inspecting its optimized parameter; for example, a pipeline-weighting term with an optimized coefficient of zero is treated as absent. The *All* column reports motif frequencies across these 100 sampled policies. We then examine two subsets of the same sample. The *Top-3 Ranked* column includes sampled policies that rank among the three lowest-cost policies within their generation, while the *Generations 8–10* column includes sampled policies generated in the final three generations. Comparing these columns provides a descriptive view of which motifs are common overall and which appear more frequently among better-performing or later-generation policies. Because the two subsets are defined from the same sample, they may overlap. Four patterns stand out:

- The discovered policies often retain inventory position but modify the associated ordering logic. The inventory-position motif appears in 77.0% of the full sample and remains similarly prevalent among top-ranked policies (75.0%) and policies from generations 8–10 (78.3%). In contrast, the standard order-up-to motif appears in only 30.0% of the full sample. Thus, AIPS frequently preserves inventory position as a useful state summary while changing how that information is translated into an order quantity.

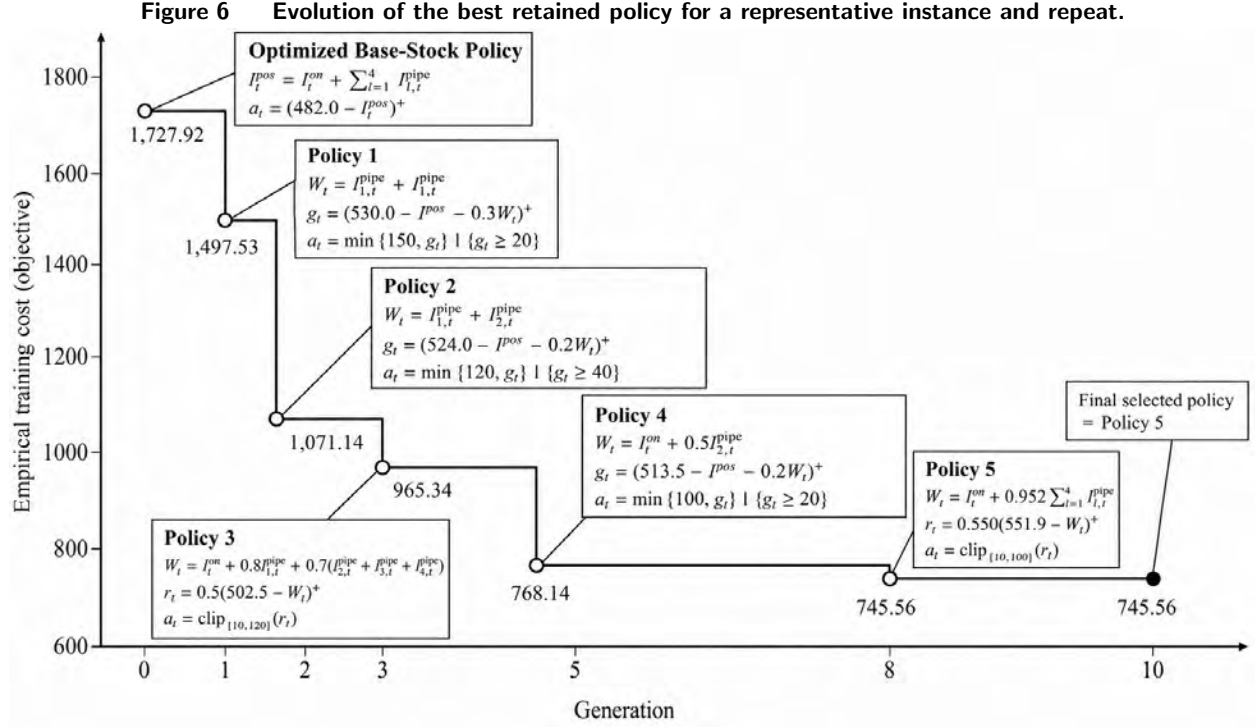
- Richer representations of pipeline inventory are common. Pipeline weighting appears in 49.0% of the full sample, while nonlinear pipeline composition appears in 28.0%. Nonlinear pipeline composition is more prevalent among top-ranked policies (35.0%) and policies from generations 8–10 (39.1%). These patterns suggest that many discovered policies distinguish among different parts of the pipeline rather than assigning equal importance to all outstanding orders. This provides one possible explanation for the lead-time results in Figure 3, where the gains from AIPS increase substantially with lead time.

- The discovered policies frequently moderate the replenishment response. Order clipping appears in 37.0% of the full sample and in 56.5% of policies from generations 8–10. Similarly, state-dependent partial adjustment appears in 33.0% of the full sample, 37.5% of top-ranked policies, and 47.8% of policies from generations 8–10. These results indicate that AIPS often retains target-seeking behavior while regulating how aggressively the policy responds to an inventory shortfall. By comparison, order smoothing and thresholding are less common, appearing in 7.0% and 13.0% of the full sample, respectively, suggesting that they serve as more specialized refinements.

- The motifs typically appear in combination rather than in isolation. A sampled policy contains 3.31 motifs on average, the median policy contains three motifs, and only 3 of the 100 sampled policies contain a single motif. Thus, the policies discovered by AIPS are best viewed as combinations of a small set of interpretable inventory-control building blocks rather than as arbitrary black-box rules or minor variations of a single benchmark policy. Overall, three recurring ingredients emerge: informative state summaries, richer representations of pipeline inventory, and moderated replenishment responses.

4.4. Representative Evolutionary Trajectory

The previous section summarizes which structural motifs recur across policies generated by AIPS, but it does not show how the search moves from one policy to another within a particular run. Figure 6 therefore traces one representative repeat for the Poisson-demand instance with lead time $L = 4$ and cost parameters $(h, p) = (1, 2)$. The horizontal axis denotes the generation, and the vertical axis reports the empirical training cost of the best policy retained up to that generation. Generation 0 corresponds to the optimized base-stock policy used to initialize the search. Whenever



AIPS discovers a policy with lower empirical training cost than the current incumbent, the figure reports the new policy's simplified functional form in an accompanying box. In this run, new policies are discovered in generations 1, 2, 3, 5, and 8, corresponding to Policies 1–5, respectively. The empirical cost decreases from 1,727.92 for the optimized base-stock benchmark to 1,497.53, 1,071.14, 965.34, 768.14, and ultimately 745.56. No better policy is found after generation 8, so Policy 5 remains the best candidate through generation 10 and is returned as the final selected policy.

To illustrate how the policy annotations in Figure 6 should be interpreted, consider Policy 1, the first policy to improve on the optimized base-stock benchmark. Since $L = 4$, denote the inventory position and the quantity arriving in the first two pipeline positions as $I_t^{\text{pos}} := I_t^{\text{on}} + \sum_{\ell=1}^4 I_{t,\ell}^{\text{pipe}}$, $W_t := I_{t,1}^{\text{pipe}} + I_{t,2}^{\text{pipe}}$. Policy 1 computes the adjusted target gap $g_t := (530.0 - I_t^{\text{pos}} - 0.3W_t)^+$, and places the order $a_t := \min\{150, g_t\} \mathbf{1}\{g_t \geq 20\}$. Relative to the base-stock policy, this rule changes both the representation of the inventory state and the response to the resulting target gap. Because I_t^{pos} already assigns unit weight to every pipeline component, subtracting an additional $0.3W_t$ gives the first two pipeline positions a total weight of 1.3, while the remaining two retain weight 1. Equivalently,

$$g_t = \left(530.0 - I_t^{\text{on}} - 1.3I_{t,1}^{\text{pipe}} - 1.3I_{t,2}^{\text{pipe}} - I_{t,3}^{\text{pipe}} - I_{t,4}^{\text{pipe}} \right)^+.$$

Thus, Policy 1 distinguishes between near- and later-arriving pipeline inventory rather than treating all outstanding orders identically. It also modifies the base-stock policy's one-step adjustment

through a threshold and an order cap:

$$a_t = \begin{cases} 0, & \text{if } g_t < 20, \\ g_t, & \text{if } 20 \leq g_t \leq 150, \\ 150, & \text{if } g_t > 150. \end{cases}$$

The threshold prevents small target gaps from triggering an order, while the upper bound limits excessively large corrective orders. Policy 1 therefore combines the inventory-position, pipeline-weighting, order-up-to, thresholding, and order-clipping motifs, and reduces the empirical training cost from 1,727.92 to 1,497.53.

Policy 2 retains essentially the same structural template as Policy 1 but uses different optimized parameter values, reducing the empirical training cost further to 1,071.14. This illustrates that two generated policies can share the same motif combination yet attain different performance because the numerical optimizer identifies different parameterizations under its finite computational budget. The subsequent incumbents introduce more substantial structural changes. Policy 3 replaces the standard inventory-position signal with a timing-sensitive discounted representation of the pipeline and combines this signal with partial adjustment and two-sided order clipping. Policy 4 returns to inventory position but augments it with a timing-weighted near-arrival signal and an upper order cap. Finally, Policy 5 uses a uniformly discounted pipeline signal together with partial adjustment and two-sided clipping. Its empirical training cost is 745.56, corresponding to a 56.9% reduction relative to the optimized base-stock benchmark. Policy 5 becomes the incumbent in generation 8 and remains the best retained policy through generation 10.

This trajectory illustrates how the structural motifs identified in Table 1 emerge within an actual search run. The search repeatedly preserves the basic target-seeking logic of inventory control while changing two key components: how the pipeline state is summarized and how aggressively the resulting target gap is converted into an order. Importantly, the evolutionary path does not simply add complexity from one generation to the next. Instead, it moves among different combinations of pipeline weighting, partial adjustment, thresholding, and clipping before arriving at a relatively parsimonious final rule.

5. Cross-Environment Generalization

This section studies whether the policy classes discovered by AIPS generalize beyond the problem instances on which they were generated. Importantly, the object transferred across instances is the policy class, not a particular policy with fixed parameter values. Specifically, for a discovered class $\Pi_\Theta := \{\pi_\theta : \theta \in \Theta\}$, we retain its functional form $\pi_\theta(\cdot)$ and feasible parameter set Θ , and re-optimize θ using training data from a new target instance that differs in its demand distribution, cost parameters, and/or lead time from the instance on which the class was discovered. We then evaluate

the resulting optimized policy against the optimized base-stock policy for that same target instance. This cross-environment test therefore measures *structural generalization*: whether a functional form discovered in one environment continues to contain high-performing policies after its parameters are recalibrated for another environment. In this way, we assess whether the structural motifs identified in §4.3 combine into reusable policy classes rather than instance-specific policies.

We use three reference instances to generate the policy classes studied in this section. All three share the same lead time and cost ratio, $L = 6$ and $R := p/h = 2$, and differ only in the demand distribution. The first has Poisson demand with mean $\mu = 100$ and standard deviation $\sigma = 10$, corresponding to a low-variability environment with coefficient of variation $\sigma/\mu = 10\%$. The second has Normal demand with $\mu = 100$ and $\sigma = 30$, representing moderate variability with coefficient of variation 30%. The third has Exponential demand with $\mu = 100$ and $\sigma = 100$, representing high variability with coefficient of variation 100%. In §5.1, we describe the three policy classes discovered from these reference instances and relate their structures to the motifs identified in §4.3. In §5.2, we define the target instances and performance metrics. In §5.3, we report the generalization results.

Table 2 Structural motifs in the three discovered policy classes.

Reference demand	Policy class	Main motifs
Poisson	Capped	Inventory position, order-up-to, order clipping
Exponential	Discounted-pipeline	Pipeline weighting, partial adjustment, thresholding
Normal	Hybrid	Pipeline weighting, partial adjustment, constant order, order clipping

5.1. Discovered Inventory Policy Classes

On the three reference instances, AIPS discovers three policy classes that we use in the generalization analysis: the capped, discounted-pipeline, and hybrid classes. We next describe their functional forms and relate them to the structural motifs identified in §4.3. For each $q \in \{C, D, H\}$, let $\Pi_{\Theta_q}^q := \{\pi_{\theta_q}^q : \theta_q \in \Theta_q\}$ denote the corresponding discovered class, where θ_q is its parameter vector and Θ_q is the feasible parameter space returned by the LLM. Table 2 summarizes the reference instance on which each class was discovered and its main structural motifs. For any state $\mathbf{I}$, let $I^{\text{pos}} := I^{\text{on}} + \sum_{k=1}^L I_k^{\text{pipe}}$ denote the inventory position, following §2. We suppress the common upper bound $\bar{a}$ on feasible order quantities because it is chosen sufficiently large to be nonbinding.

Capped policy class. The class discovered on the Poisson reference instance preserves the inventory-position representation and target-gap logic of the base-stock policy, but caps the resulting order quantity. Let $\theta_C := (\theta_C, \kappa_C) \in \Theta_C$, where θ_C is the target inventory position and κ_C is the order cap. For any state $\mathbf{I}$, the policy is $\pi_{\theta_C}^C(\mathbf{I}) := \min\{\kappa_C, (\theta_C - I^{\text{pos}})^+\}$. Thus, relative to the base-stock class, the state representation is unchanged; the structural modification is the order-clipping motif, which prevents a large target gap from generating an equally large corrective order. When κ_C is nonbinding, the policy reduces to the standard base-stock rule.

Discounted-pipeline policy class. The class discovered on the Exponential reference instance modifies both how pipeline inventory is represented and how an inventory shortfall is translated into an order. Let $\theta_D := (\theta_D, \omega_D, \rho_D, \tau_D) \in \Theta_D$, and define $x_D(\mathbf{I}) := I^{\text{on}} + \omega_D \sum_{k=1}^L I_k^{\text{pipe}}$ and $g_D(\mathbf{I}) := \theta_D - x_D(\mathbf{I})$. For any state $\mathbf{I}$, the policy is $\pi_{\theta_D}^D(\mathbf{I}) := \rho_D(g_D(\mathbf{I}))^+ \mathbf{1}\{g_D(\mathbf{I}) > \tau_D\}$. Here, ω_D determines the weight assigned to pipeline inventory, ρ_D determines the fraction of the target gap that is replenished, and τ_D creates an inaction region for small shortfalls. Thus, relative to the base-stock class, the discounted-pipeline class modifies both the state representation and the replenishment response through the pipeline-weighting, partial-adjustment, and thresholding motifs.

Hybrid policy class. The class discovered on the Normal reference instance combines position-specific pipeline weighting with a moderated replenishment response and a state-independent baseline component. Let $\theta_H := (\theta_H, \boldsymbol{\omega}, \alpha, \boldsymbol{\beta}, \rho_H, \delta_H, \underline{a}_H) \in \Theta_H$, where $\boldsymbol{\omega} = (\omega_1, \dots, \omega_L)$ and $\boldsymbol{\beta} = (\beta_1, \dots, \beta_L)$. Define the effective weight on pipeline position k as $v_k := \omega_k - \alpha\beta_k$ and the corresponding inventory signal as $x_H(\mathbf{I}) := I^{\text{on}} + \sum_{k=1}^L v_k I_k^{\text{pipe}}$. For any state $\mathbf{I}$, the policy is $\pi_{\theta_H}^H(\mathbf{I}) := \max\{\underline{a}_H, \rho_H(\theta_H - x_H(\mathbf{I})) + (1 - \rho_H)\delta_H\}$. The effective weights v_k allow different pipeline positions to receive different credits, ρ_H moderates the response to the target gap, $(1 - \rho_H)\delta_H$ provides a state-independent replenishment component, and $\underline{a}_H$ imposes a lower order clip. Thus, the hybrid class combines the pipeline-weighting, partial-adjustment, constant-order, and order-clipping motifs within a single replenishment rule.

Taken together, the three classes illustrate different ways in which AIPS recombines the structural building blocks identified in §4.3. The capped class preserves the base-stock state representation and modifies only the replenishment response, whereas the discounted-pipeline and hybrid classes modify both how pipeline inventory is represented and how the resulting inventory signal is translated into an order. These functional forms, rather than their parameter values on the reference instances, are the objects transferred in the experiments below: for each target environment, we retain the class structure and reoptimize its parameters before evaluating performance.

5.2. Target Instances and Performance Metrics

We evaluate whether the three policy classes discovered on the reference instances remain effective across a broad collection of target environments. Each target instance k is characterized by a demand family $\mathcal{F}_k$, mean μ_k , standard deviation σ_k , lead time L_k , and lost-sales-penalty-to-holding-cost ratio $R_k := p_k/h_k$. We consider 13 demand families spanning bounded continuous distributions (Beta, continuous Uniform, and Triangular), bounded discrete distributions (Binomial and discrete Uniform), light- to moderate-tailed unbounded distributions (Gamma, Geometric, Negative Binomial, Normal, and Weibull), and heavy-tailed or highly skewed distributions (Lognormal, Pareto, and zero-inflated Negative Binomial). Varying the distribution parameters produces 629 distinct demand

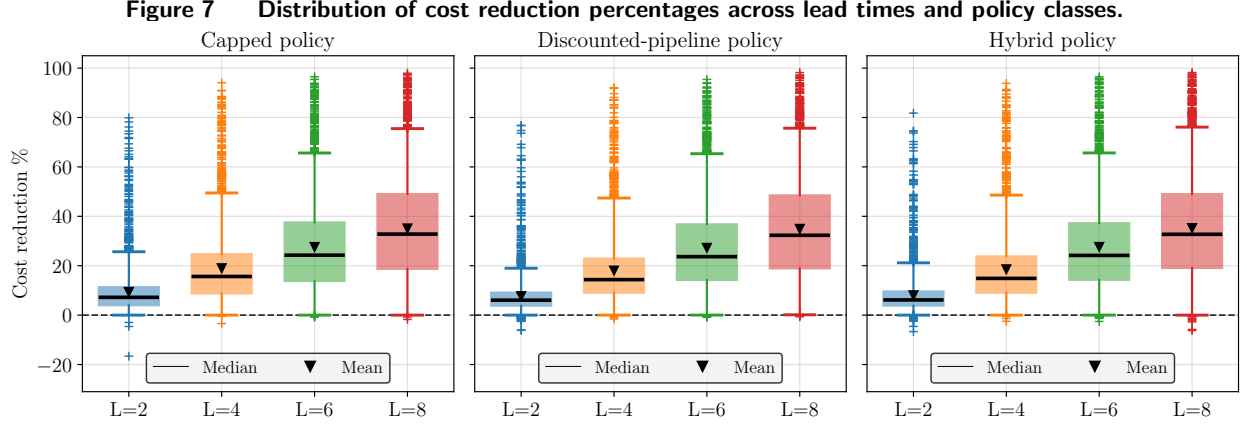
specifications $(\mathcal{F}, \mu, \sigma)$; the complete construction is reported in Appendix EC.3. Combining these specifications with $L \in \{2, 4, 6, 8\}$ and $R \in \{2, 4, 6, 10\}$ yields $629 \times 4 \times 4 = 10,064$ target instances.

For each target instance, we transfer the three policy classes described in §5.1 while reoptimizing their parameters for the new environment. Specifically, for each policy class $q \in \{C, D, H\}$, the functional form and feasible parameter space $\Pi_{\Theta_q}^q := \{\pi_{\theta_q}^q : \theta_q \in \Theta_q\}$ are held fixed, and the numerical optimization procedure in §3.2 is applied to obtain an instance-specific parameter vector $\hat{\theta}_{k,q}$ for target instance k . We similarly reoptimize the base-stock parameter to obtain $\hat{\theta}_{k,B}$. Thus, the four policies compared on instance k are $\pi_{\hat{\theta}_{k,C}}^C$, $\pi_{\hat{\theta}_{k,D}}^D$, $\pi_{\hat{\theta}_{k,H}}^H$, and $\pi_{\hat{\theta}_{k,B}}^{BS}$. Thus, our experiments evaluate the structural generalization: whether a class discovered in one environment continues to contain effective policies when the demand distribution, lead time, or cost ratio changes.

Following §4, we use percentage cost reduction relative to the optimized base-stock policy as the primary performance metric. For each target instance k and policy class q , let CR_k^q denote the percentage reduction in empirical cost achieved by the re-optimized policy from class q relative to the optimized base-stock policy on the same instance. Thus, $CR_k^q > 0$ indicates that class q yields a lower-cost policy than the base-stock benchmark, with larger values indicating greater improvement. To compare policy classes across the $K = 10,064$ target instances, we examine both the mean cost reduction and the distribution of CR_k^q across instances. As a complementary measure, we rank the four optimized policies—capped, discounted-pipeline, hybrid, and base-stock—within each target instance according to empirical cost. Let $PR_k^q \in \{1, 2, 3, 4\}$ denote the policy rank, where rank 1 corresponds to the lowest empirical cost and rank 4 to the highest. We similarly report the mean policy rank and its distribution across target instances. Thus, lower values indicate better relative performance, with a rank close to 1 meaning that the policy is typically the best-performing of the four alternatives.

5.3. Results

We begin with the aggregate performance across all 10,064 target instances. The capped, discounted-pipeline, and hybrid classes achieve mean cost reductions of 22.60%, 21.75%, and 22.10%, respectively, relative to the optimized base-stock benchmark. Their corresponding mean policy ranks are 1.93, 2.17, and 2.01, compared with 3.89 for the base-stock policy. Thus, all three discovered classes generalize strongly across environments that differ from the reference instances: after reoptimizing their parameters for each target instance, they reduce cost by more than 21% on average and typically rank well ahead of the base-stock benchmark. Among the three discovered classes, the capped class performs best on average, with the highest mean cost reduction and lowest mean policy rank, followed closely by the hybrid and discounted-pipeline classes. More broadly, the key result is not that a single discovered class dominates across all environments, but that each transferred functional form continues to contain high-performing policies across a wide range of demand distributions, lead times, and cost ratios.



Impact of lead time variation. Figure 7 shows how the distribution of cost reduction changes with lead time for the capped, discounted-pipeline, and hybrid policy classes. The left, middle, and right panels correspond to the three classes, respectively. In each panel, the vertical axis reports cost reduction relative to the optimized base-stock policy, and the boxplots correspond to $L \in \{2, 4, 6, 8\}$. The solid line denotes the median and the triangle denotes the mean. Across all three classes, the distributions shift markedly upward as lead time increases. For the capped class, mean cost reduction rises from 9.20% at $L = 2$ to 18.86%, 27.43%, and 34.90% at $L = 4, 6, 8$, respectively. The corresponding values are 7.47%, 17.84%, 27.00%, and 34.70% for the discounted-pipeline class, and 7.80%, 18.32%, 27.32%, and 34.94% for the hybrid class. Moreover, Table 3 provides a complementary comparison based on relative ranking. All three discovered policy classes rank substantially better than the base-stock benchmark, whose mean policy rank remains close to 4 for every lead time. Among the discovered classes, the capped class ranks best at $L = 2$ and $L = 4$, while the hybrid class has the lowest mean policy rank at $L = 6$ and $L = 8$. Thus, although all three classes perform similarly at longer lead times, the hybrid class gains a slight relative advantage as the pipeline lengthens.

Table 3 Mean policy rank across lead times.

L	Capped	Discounted-pipeline	Hybrid	Base-stock
2	1.78	2.24	2.12	3.86
4	1.87	2.24	1.99	3.89
6	1.99	2.13	1.98	3.90
8	2.06	2.07	1.95	3.91

Together, Figure 7 and Table 3 show that the transferred policy classes substantially outperform the optimized base-stock benchmark across the full lead-time range, with the advantage becoming especially pronounced for long lead times. The upward shift in the full cost-reduction distributions indicates that this pattern is not driven by only a few favorable target instances. One interpretation is that, as lead time increases, the base-stock policy's aggregation of all pipeline inventory into a single inventory-position measure becomes increasingly restrictive, creating greater value for policy classes that represent the pipeline more flexibly and moderate the replenishment response.

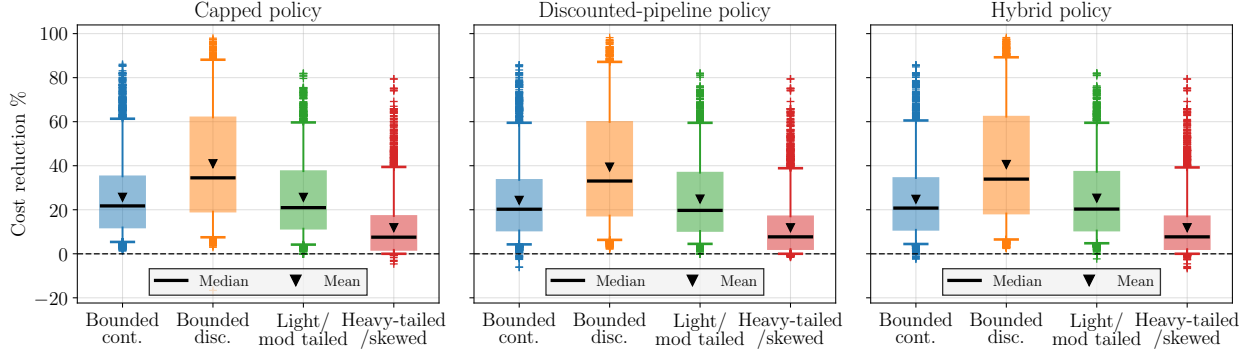
Table 4 Mean cost reduction and mean policy rank across lost-sales-cost-to-holding-cost ratios $R = p/h$.

R	Capped		Discounted-pipeline		Hybrid		Base-stock	
	Cost reduction	Policy rank	Cost reduction	Policy rank	Cost reduction	Policy rank	Cost reduction	Policy rank
2	26.93	2.22	26.80	2.04	26.95	1.90	–	3.84
4	23.35	1.98	22.84	2.14	23.11	1.98	–	3.91
6	21.30	1.82	20.29	2.24	20.69	2.05	–	3.89
10	18.81	1.68	17.08	2.27	17.63	2.12	–	3.92

Impact of cost ratio. Table 4 reports mean cost reduction and mean policy rank for each policy class across the lost-sales-cost-to-holding-cost ratios $R \in \{2, 4, 6, 10\}$. For all three discovered classes, mean cost reduction decreases monotonically as R increases. The capped class declines from 26.93% at $R = 2$ to 18.81% at $R = 10$; the discounted-pipeline class declines from 26.80% to 17.08%; and the hybrid class declines from 26.95% to 17.63%. Thus, the advantage over the optimized base-stock benchmark is largest when lost sales are relatively inexpensive compared with holding inventory and narrows as R increases. The mean policy rank results tell the same broad story: the base-stock benchmark remains close to rank 4, with values between 3.84 and 3.92, while the discovered classes remain near rank 2. Among the discovered classes, the hybrid class performs best at $R = 2$, the capped and hybrid classes tie at $R = 4$, and the capped class has the lowest mean policy rank at $R = 6$ and $R = 10$.

The key insight is that the transferred policy classes remain substantially better than the optimized base-stock benchmark across the full range of cost ratios, although the size of the improvement declines as lost sales become more expensive. One interpretation is that a higher R makes the base-stock policy’s aggressive protection against stockouts more competitive, leaving less room for improvement. At the same time, the shift in relative performance across discovered classes is consistent with their structural differences: the hybrid class performs particularly well when holding inventory is relatively costly, whereas the capped class becomes relatively stronger as the lost-sales penalty increases. These patterns suggest that different combinations of the motifs identified in §4.3 are better suited to different cost regimes.

Impact of demand distribution. Figure 8 shows how cost reduction varies across four broad demand groups: bounded continuous, bounded discrete, light- to moderate-tailed, and heavy-tailed or highly skewed distributions. The left, middle, and right panels correspond to the capped, discounted-pipeline, and hybrid policy classes, respectively. In each panel, the vertical axis reports percentage cost reduction relative to the optimized base-stock policy, while the solid line and triangle denote the median and mean. All three classes exhibit a similar pattern across demand environments. Cost reductions are largest under bounded discrete demand, remain substantial under bounded continuous and light- to moderate-tailed demand, and are markedly smaller under heavy-tailed or highly skewed demand. The latter group also contains observations near or below

Figure 8 Distribution of cost reduction percentages across demand families and policy classes.

zero for all three classes, indicating that the advantage over base-stock is less consistent in these environments. Overall, the dominant variation is across demand environments rather than across the three transferred classes. These results suggest that the discovered structures transfer particularly well to bounded and moderately behaved demand settings, whereas heavy tails and strong skewness make their performance gains less uniform across target instances.

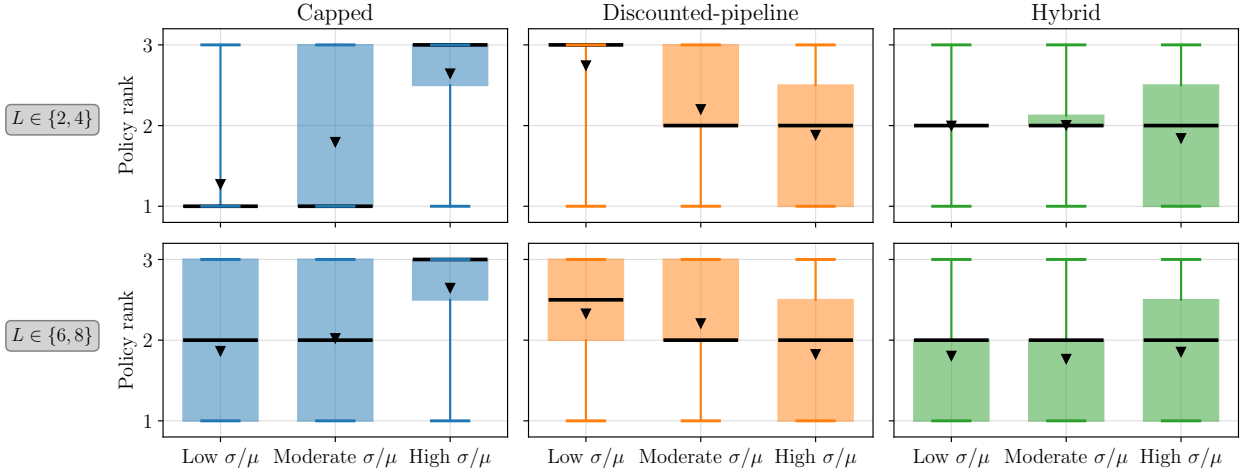
Figure 9 Distribution of policy ranks across demand-variability and lead-time regimes.

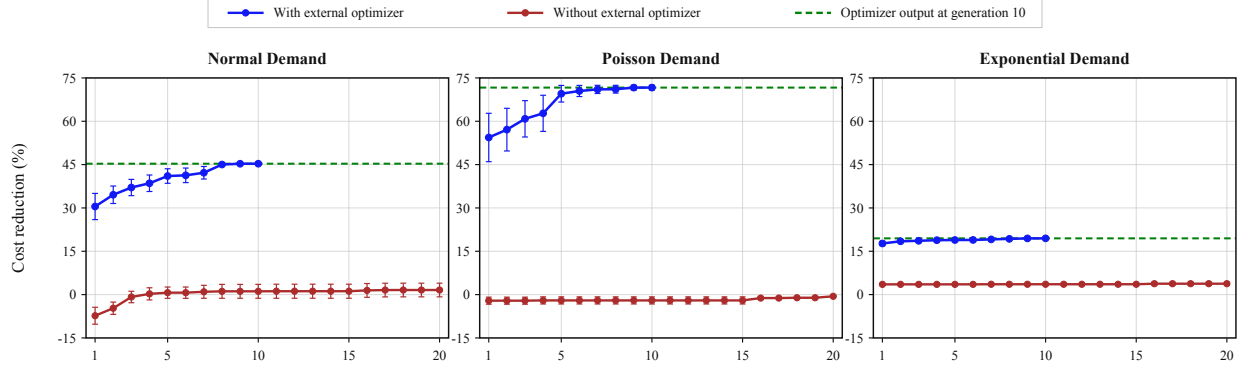
Figure 9 provides a more direct comparison among the three discovered classes across demand variability and lead time. The columns correspond to the capped, discounted-pipeline, and hybrid classes, while the top and bottom rows correspond to short lead times $L \in \{2, 4\}$ and long lead times $L \in \{6, 8\}$, respectively. Within each panel, the horizontal axis distinguishes low, moderate, and high coefficients of variation σ/μ , and the vertical axis reports policy rank among the three discovered classes, with lower values indicating better relative performance. The capped class performs particularly well under low variability, especially at short lead times, where its median rank is 1, but its relative performance deteriorates as variability increases. In contrast, the discounted-pipeline and hybrid classes become more competitive under high variability and substantially outperform the capped class in relative rank. The hybrid class also becomes relatively stronger as lead time increases, particularly under moderate variability.

Taken together, the two figures show that no single transferred class is uniformly best. When demand is relatively stable, retaining the standard inventory-position representation while limiting large corrective orders, as in the capped class, is often sufficient. As variability increases, richer treatment of pipeline inventory becomes more valuable, favoring the discounted-pipeline and hybrid classes. Longer lead times further increase the value of distinguishing pipeline information and moderating the replenishment response. Thus, the relative performance of the discovered classes varies systematically with the demand environment and pipeline length, providing evidence that different combinations of structural motifs are useful in different operating regimes.

6. Value of Optimization

The previous sections use AIPS with an external numerical optimizer throughout the evolutionary search. Recall that for every LLM-generated class Π_Θ , the optimizer searches over Θ to obtain a low-cost policy $\hat{\pi}$; this optimized policy, together with its parameter vector and empirical cost, then enters survival selection and may subsequently serve as a parent. Optimization can therefore affect AIPS through two distinct channels. First, it can improve a generated class directly by calibrating its parameters. Second, because the optimized policy is used to evaluate the class and guide subsequent LLM proposals, it can change the evolutionary search path itself. In particular, a useful functional form with poorly calibrated LLM-supplied parameters may appear unattractive without optimization and be discarded before it can influence later generations. This section separates these two effects and asks whether optimization is valuable only for parameter calibration or also for guiding policy-class search.

Experimental comparison. We compare two variants of AIPS while holding the LLM backbone and the remaining search procedure fixed. The *optimizer-throughout* variant is the default implementation used in §§4–5: for every feasible LLM proposal (Π_Θ, θ_0) , the external optimizer searches within Θ , and the resulting optimized policy is evaluated and used for selection and feedback. In the *no-optimizer* variant, this within-class optimization step is removed. The policy instantiated at the LLM-supplied parameter vector θ_0 is evaluated directly, and its functional form, raw parameter values, and empirical cost provide the feedback for subsequent generations. Thus, the two variants differ precisely in whether policy classes are evaluated after parameter optimization or at their LLM-supplied initialization. Regarding the demand instances, we use a subset of instances introduced in §4: Normal with standard deviation 30, Poisson, and Exponential, with a fixed $L = 6$ and cost parameters $(h, p) = (1, 2)$. We use the cost-reduction metric defined in §4. Because the no-optimizer search improves more slowly, we allow it to run for 20 generations, twice the default 10-generation budget of the optimizer-throughout variant. This gives the no-optimizer variant additional opportunity to compensate for the absence of numerical parameter search.

Figure 10 Evolution of average cost reduction with and without external parameter optimization.

Performance level and search trajectory. Figure 10 compares the evolutionary trajectories under the two variants using DeepSeek V3, grouped by the Poisson, Normal, and Exponential demand environments. In all three panels, the horizontal axis denotes generation and the vertical axis reports average cost reduction relative to the optimized base-stock policy. The dotted line with diamonds corresponds to optimizer-throughout, while the solid line with circles corresponds to no-optimizer. As shown across the three distribution types, the difference is large from the start and persists throughout the search. With optimization, the average cost reduction is already positive in generation 1 and continues to rise by generation 10. Without optimization, the initial LLM-generated policies perform substantially worse than the optimized base-stock benchmark, and the search requires several generations merely to reach positive average improvement. Additional generations help, but do not close the gap: even after 20 generations, the no-optimizer variant is still well below the optimizer-throughout result after 10 generations. Thus, optimization does more than provide a small local refinement to LLM-generated parameter values. It places the evolutionary search on a substantially stronger performance trajectory from its earliest generations, and doubling the number of LLM search generations does not compensate for removing the numerical optimizer.

Table 5 Convergence speed with and without external optimization.

	Median	Mean	Interquartile range	Range
With optimizer	7.5	6.8	[5,9]	[1,10]
Without optimizer	15.0	11.6	[4,18]	[1,20]

Convergence speed. Table 5 examines whether optimization also accelerates convergence. For each run, we record the first generation at which the incumbent reaches within 1% of that variant’s own final performance. Because this target is defined separately for each variant, the measure abstracts from differences in final performance and captures only how quickly each search approaches its own eventual level across all instances. With optimization, the median convergence time is 7.5 generations, compared with 15.0 generations without optimization; the corresponding

means are 6.8 and 11.6 generations. The interquartile range is also substantially narrower with optimization, [5, 9] versus [4, 18]. Thus, the external optimizer not only improves the quality of the policies discovered but also allows the evolutionary search to approach its eventual performance substantially faster and more consistently.

Separating parameter calibration from search guidance. The preceding comparisons establish that optimizer-throughout performs better and converges faster, but they do not yet identify why. One possibility is purely parametric: perhaps the no-optimizer search discovers equally useful policy classes, but performs poorly because the LLM assigns bad parameter values to those classes. If so, taking the class underlying the final no-optimizer policy and optimizing its parameters *after* the evolutionary search should recover most of the performance gap. The alternative is a search-guidance effect: evaluating poorly calibrated policies during the search changes which classes survive, which policies become parents, and consequently which structural motifs the LLM explores in later generations. In that case, optimizing only the final no-optimizer class cannot recover the classes that were discarded or never reached because earlier feedback was less informative.

To distinguish these two effects, for each no-optimizer run we take the policy class underlying its final retained policy and optimize its parameters once after the evolutionary search is complete. Let π_k^{with} denote the policy returned by optimizer-throughout, π_k^{no} the policy returned by the no-optimizer search, and π_k^{post} the policy obtained by applying final-step parameter optimization to the class underlying π_k^{no} . We define the total performance gap as the difference between the no-optimizer and optimizer-throughout policies, normalized by the optimized base-stock cost:

$$G_k^{\text{total}} := 100 \frac{\hat{J}_k(\pi_k^{\text{no}}) - \hat{J}_k(\pi_k^{\text{with}})}{\hat{J}_k(\pi_{k,B})}.$$

Thus, G_k^{total} measures how much performance is lost when optimization is removed from the evolutionary search. We then separate this loss into two components. The first, G_k^{param} , is the portion recovered by optimizing the parameters of the final class discovered without optimization; the second, G_k^{search} , is the gap that remains after this final-step optimization:

$$G_k^{\text{param}} := 100 \frac{\hat{J}_k(\pi_k^{\text{no}}) - \hat{J}_k(\pi_k^{\text{post}})}{\hat{J}_k(\pi_{k,B})}, \quad G_k^{\text{search}} := 100 \frac{\hat{J}_k(\pi_k^{\text{post}}) - \hat{J}_k(\pi_k^{\text{with}})}{\hat{J}_k(\pi_{k,B})}.$$

By construction, $G_k^{\text{total}} = G_k^{\text{param}} + G_k^{\text{search}}$. A large G_k^{param} would indicate that the benefit of optimization comes mainly from better parameter calibration within the final discovered class. In contrast, a large G_k^{search} indicates that final-step calibration cannot recover most of the advantage of optimizer-throughout, consistent with optimization also improving the policy classes reached through the evolutionary search.

Table 6 Decomposition of the test-performance gap between optimizer-throughout and no-optimizer search.

Demand	Total gap	Gap closed by final-step optimization	Remaining gap
Poisson	72.9	1.1	71.8
Exponential	16.1	0.0	16.1
Normal	43.8	3.1	40.7

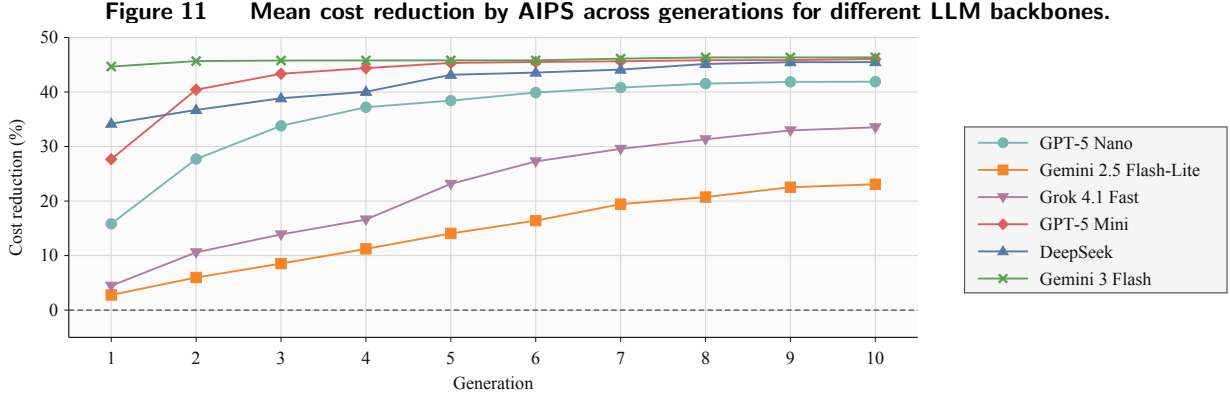
Table 6 shows that final-step parameter optimization closes only a small fraction of the performance gap. For Poisson demand, only 1.1 percentage points of the 72.9-point gap are recovered; for Exponential demand, essentially none of the 16.1-point gap is recovered; and for Normal demand, only 3.1 of 43.8 percentage points are recovered. Thus, the large majority of the advantage from optimizer-throughout remains even after the final no-optimizer class is numerically calibrated. This decomposition provides the clearest evidence that the value of optimization extends beyond parameter tuning. By evaluating each proposed class after calibration, the optimizer provides more informative feedback for survival and parent selection throughout the evolutionary process. A promising class with poor LLM-supplied parameters can therefore remain in the search and influence subsequent proposals. Final-step optimization can improve the parameters of the class that eventually survives the no-optimizer search, but it cannot recover promising classes that were discarded earlier or the different evolutionary path that optimization-guided feedback would have produced.

Overall, the experiments identify three roles for the external optimizer in AIPS. It improves within-class parameter calibration, accelerates convergence, and, most importantly, improves the feedback that guides policy-class search. The last effect explains why simply adding optimization after the evolutionary search recovers only a small portion of the performance gap. It also reinforces the interpretation of §5: the discovered functional forms can transfer across environments, but their parameters must be recalibrated to realize their value in a new environment. The next section examines whether these findings are robust across different LLM backbones.

7. LLM Backbone Ablation

The previous sections evaluate AIPS using DeepSeek V3 as the LLM backbone. We now examine how sensitive the search is to this choice by comparing six backbones: GPT-5 Nano, Gemini 2.5 Flash-Lite, Grok 4.1 Fast Non-Reasoning, GPT-5 Mini, DeepSeek V3, and Gemini 3 Flash. We use the same demand instances as in §6. Our goal is not to rank these models as general-purpose LLMs, but to assess their effectiveness as policy-class generators within AIPS. A stronger backbone should propose useful executable policy structures and respond effectively to evolutionary feedback. We compare the resulting searches using mean cost reduction relative to the optimized base-stock benchmark.

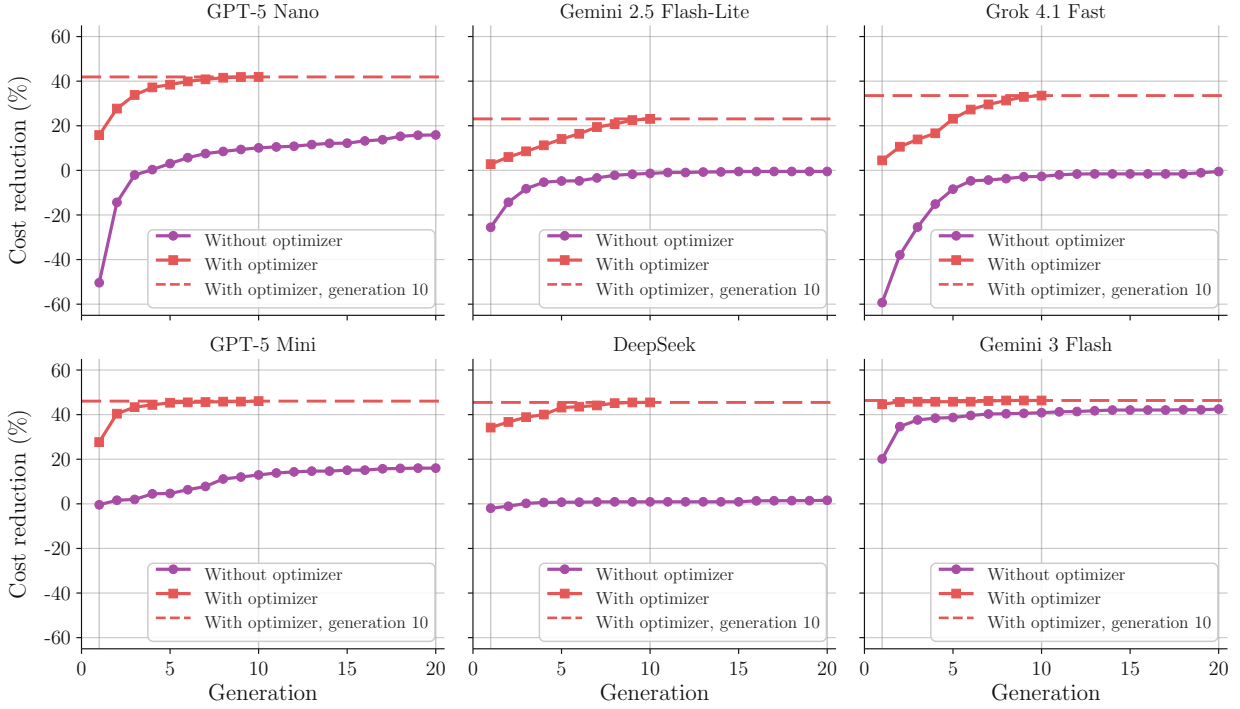
Impact of the LLM backbone. Figure 11 reports the mean cost reduction achieved by AIPS across generations for each LLM backbone, with the external optimizer enabled. The horizontal



axis denotes generation, and the vertical axis reports mean cost reduction relative to the optimized base-stock policy; the dashed horizontal line at zero marks the base-stock benchmark. Performance varies substantially across backbones. Gemini 3 Flash is strongest from the outset, achieving about 45% mean cost reduction in generation 1 and remaining around 46% thereafter. GPT-5 Mini rises from roughly 28% in generation 1 to above 40% by generation 2 and approaches 46% by generation 10. DeepSeek V3 also performs strongly, rising from about 34% in generation 1 to approximately 45% by generation 10. GPT-5 Nano improves more gradually, from about 16% to 42%. Grok 4.1 Fast Non-Reasoning improves steadily, reaching approximately 34% by the final generation. Gemini 2.5 Flash-Lite is the weakest backbone in this experiment, although its mean cost reduction still increases from about 3% in generation 1 to 23% in generation 10.

The key insight is that the choice of LLM backbone materially affects both the speed and the level of improvement achieved by AIPS. Because the same external optimizer is used across backbones, these differences point to variation in the quality of the policy classes supplied by the LLM. Stronger backbones such as Gemini 3 Flash, GPT-5 Mini, and Deepseek V3 generate high-performing, optimizable structures early, whereas others require more generations or reach lower performance levels. At the same time, the steady improvement of GPT-5 Nano and Grok 4.1 Fast Non-Reasoning shows that evolutionary feedback can progressively improve the structural search even when the strongest policy classes are not identified immediately.

Interaction between the LLM backbone and optimization. Figure 12 examines how the value of the external optimizer varies across LLM backbones. Each panel corresponds to one backbone, with generation on the horizontal axis and mean cost reduction relative to the optimized base-stock policy on the vertical axis. The red squares show AIPS with the optimizer over the default 10 generations, while the purple circles show the no-optimizer variant, which is allowed to continue for 20 generations. The dashed horizontal line marks the cost reduction attained by the optimizer-enabled variant at generation 10. Thus, the figure shows both the immediate performance effect of optimization and whether additional LLM generations can compensate for its absence.

Figure 12 Cost reduction across LLM backbones and generations with and without the external optimizer.

Two patterns stand out. First, optimization improves performance substantially for every backbone and generally does so much earlier in the search. For GPT-5 Nano, mean cost reduction reaches about 42% with optimization by generation 10, compared with only about 16% without optimization after 20 generations. The corresponding comparison is approximately 23% versus 0% for Gemini 2.5 Flash-Lite, 34% versus 0% for Grok 4.1 Fast Non-Reasoning, 46% versus 16% for GPT-5 Mini, and 45% versus 2% for DeepSeek V3. In each of these cases, doubling the number of generations does not compensate for removing the optimizer. Second, the magnitude of this effect depends strongly on the backbone. Gemini 3 Flash is the notable exception: even without optimization, its mean cost reduction rises from about 20% in generation 1 to above 40% by later generations, approaching the roughly 46% attained with optimization. Thus, Gemini 3 Flash appears to supply both strong policy structures and relatively effective initial parameter values, whereas for backbones such as GPT-5 Mini and GPT-5 Nano, optimization is especially important for realizing the value of the generated classes.

Together, Figures 11 and 12 clarify the complementary roles of the LLM and the optimizer in AIPS. The LLM determines which policy classes enter the search, while the optimizer reveals how well those classes can perform after parameter calibration. Optimization, therefore, improves every backbone, but it cannot fully eliminate differences in structural-search quality across LLMs.

Moreover, Table EC.4, which complements Figure 12, reports the generation-by-generation cost reductions with and without the optimizer, together with their standard deviations for each backbone. Table EC.5 provides a corresponding out-of-sample comparison at the termination of each

search—generation 20 without the optimizer and generation 10 with the optimizer—reporting the mean, standard deviation, minimum, and maximum test-set cost reduction for each backbone. The test-set results exhibit the same broad performance patterns observed in the in-sample trajectories.

Table 7 Observed LLM expense per repeat across backbones.

LLM backbone	Gemini 2.5 Flash-Lite	GPT-5 Nano	Grok 4.1 Fast Non-Reasoning	DeepSeek V3 0324	GPT-5 Mini	Gemini 3 Flash
Expense (USD)	\$0.087	\$0.101	\$0.192	\$0.243	\$0.299	\$0.505

Notes: The reported values are the observed LLM expenses for one repeat under the common experimental configuration in §4. They should therefore be interpreted as experiment-specific usage expenses rather than standalone model prices.

Expense–performance trade-off. The LLM backbones also differ substantially in their observed usage expenses. Table 7 shows that the expense of one repeat ranges from approximately \$0.087 for Gemini 2.5 Flash-Lite to \$0.505 for Gemini 3 Flash, a nearly sixfold difference. Despite this variation, all six backbones produce positive mean cost reductions relative to the optimized base-stock benchmark when combined with external optimization. Search performance, however, is not determined by LLM expense alone. The more expensive backbones, particularly Gemini 3 Flash, GPT-5 Mini, and Deepseek V3 tend to identify strong policy classes quickly and attain large cost reductions within relatively few generations. Yet less expensive backbones can also perform well. GPT-5 Nano, for example, has an observed expense of only \$0.101 per repeat but eventually achieves mean cost reductions above 40%. Grok 4.1 Fast Non-Reasoning occupies an intermediate expense–performance region, while Gemini 2.5 Flash-Lite is the least expensive but also produces the smallest improvements.

These results show that the choice of LLM backbone affects both the quality and speed of policy-class search. Stronger backbones tend to identify high-performing policy classes earlier, although less expensive models can also perform well when combined with external optimization. More importantly, the external optimizer improves performance across all six backbones, but does not eliminate differences in structural-search quality. Thus, AIPS does not depend on a particular LLM backbone, while improvements in LLM capability can directly translate into better and faster policy discovery.

8. Conclusion

In this paper, we study whether LLMs, numerical optimization, and data-driven evaluation can be combined to automate the design of inventory policy classes. We propose AIPS, an evolutionary framework in which an LLM generates parameterized replenishment-policy classes, a numerical optimizer searches for strong policies within each class, and the resulting optimized policies and

their empirical performance guide subsequent generations of structural search. Using lost-sales inventory systems with positive lead times as a test bed, we find that AIPS discovers policy classes that substantially outperform the optimized base-stock benchmark. The resulting policies are also interpretable: rather than behaving as opaque black-box rules, they repeatedly combine recognizable inventory-control motifs, including richer representations of pipeline inventory, partial adjustment, clipping, thresholding, and state-dependent replenishment.

These discovered structures also generalize beyond the instances on which they are generated. After retaining their functional forms and feasible parameter spaces and reoptimizing their parameters for new environments, the capped, discounted-pipeline, and hybrid classes achieve mean cost reductions exceeding 21% across more than 10,000 target instances spanning different demand distributions, lead times, and cost ratios. Their relative performance varies systematically across operating regimes, indicating that different motif combinations are useful under different inventory conditions. Our ablation experiments further show that numerical optimization is central to the search process: it not only calibrates parameters and accelerates convergence, but also improves the feedback used to select and evolve policy classes. Consistent with this mechanism, applying optimization only after the search recovers only a small portion of the performance lost when optimization is removed from the evolutionary loop. Comparisons across six LLM backbones also show that the quality of structural search depends on the LLM, while optimization improves realized performance across all backbones.

More broadly, our findings suggest that LLMs can complement domain-specific knowledge about effective policies. In particular, AIPS can take such expert-designed policies as starting points and systematically build on them by adding, combining, and refining structural motifs, potentially producing stronger policy classes than the original domain-based designs. Future work could develop richer mechanisms for representing and recombining structural motifs and extend the framework to settings with nonstationary demand, stochastic lead times, capacity constraints, multi-item systems, and multi-echelon networks.

References

- Adelman, Daniel, Christiane Barz, Alba V. Olivares-Nadal. 2025. Dynamic basis function generation for network revenue management. *INFORMS Journal on Computing* **0**(0).
- Arrow, Kenneth J., Theodore Harris, Jacob Marschak. 1951. Optimal inventory policy. *Econometrica* **19**(3) 250–272.
- Baek, Jackie. 2026. Llms can design near-optimal OR algorithms. *Working Paper* **0**(0).
- Baek, Jackie, Yaopeng Fu, Will Ma, Tianyi Peng. 2026. AI agents for inventory control: Human-LLM-OR complementarity. *Working Paper* **0**(0).

- Ban, Gah-Yi, Cynthia Rudin. 2019. The big data newsvendor: Practical insights from machine learning. *Operations Research* **67**(1) 90–108.
- Chen, Hao, Gonzalo Esteban Constante-Flores, Krishna Sri Ipsit Mantri, Sai Madhukiran Kompalli, Akshdeep Singh Ahluwalia, Can Li. 2026. OptiChat: Bridging optimization models and practitioners with large language models. *INFORMS Journal on Data Science* **5**(3) 199–220.
- Cohen, Maxime C., Tinglong Dai, Georgia Perakis, Narendra Agrawal, Gad Allon, Robert N. Boute, Gérard P. Cachon, Zhe Chen, Morris A. Cohen, Rares Cristian, Vinayak Deshpande, et al. 2026. OM forum—supply chain management in the AI era: A vision statement from the operations management community. *Manufacturing & Service Operations Management* **28**(3) 687–705.
- Csaszar, Felipe A., Harsh Ketkar, Hyunjin Kim. 2024. Artificial intelligence and strategic decision-making: Evidence from entrepreneurs and investors. *Strategy Science* **9**(4) 322–345.
- DeHoratius, Nicole, Adam J. Mersereau, Linus Schrage. 2008. Retail inventory management when records are inaccurate. *Manufacturing & Service Operations Management* **10**(2) 257–277.
- Ding, Jingying, Woonghee Tim Huh, Ying Rong. 2024. Feature-based inventory control with censored demand. *Manufacturing & Service Operations Management* **26**(3) 1157–1172.
- Fildes, Robert, Paul Goodwin, Michael Lawrence, Konstantinos Nikolopoulos. 2009. Effective forecasting and judgmental adjustments: An empirical evaluation and strategies for improvement in supply-chain planning. *International Journal of Forecasting* **25**(1) 3–23.
- Goldberg, David A., Dmitriy A. Katz-Rogozhnikov, Yingdong Lu, Mayank Sharma, Mark S. Squillante. 2016. Asymptotic optimality of constant-order policies for lost sales inventory models with large lead times. *Mathematics of Operations Research* **41**(3) 898–913.
- Huang, Chenyu, Jianghao Lin, Zhengyang Tang, Bo Jiang, Ruoqing Jiang, Benyou Wang, Lai Wei. 2026. InvEvolve: Evolving white-box inventory policies via large language models with performance guarantees. *Working paper* **0**(0).
- Huh, Woonghee Tim, Ganesh Janakiraman, John A. Muckstadt, Paat Rusmevichientong. 2009. Asymptotic optimality of order-up-to policies in lost sales inventory systems. *Management Science* **55**(3) 404–420.
- Huh, Woonghee Tim, Retsef Levi, Paat Rusmevichientong, James B. Orlin. 2011. Adaptive data-driven inventory control with censored demand based on Kaplan-Meier estimator. *Operations Research* **59**(4) 929–941.
- Iyer, Ananth V, Linus E Schrage. 1992. Analysis of the deterministic (s, S) inventory problem. *Management Science* **38**(9) 1299–1313.
- Jiang, Juyong, Fan Wang, Jiasi Shen, Sungju Kim, Sunghun Kim. 2026. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology* **35**(2) 1–72.
- Lagzi, Saman, Ningyuan Chen, Joseph Milner. 2026. Using neural networks to guide data-driven operational decisions. *Management Science* **0**(0).

-
- Lin, Meichun, Woonghee Tim Huh, Harish Krishnan, Joline Uichanco. 2022. Data-driven newsvendor problem: Performance of the sample average approximation. *Operations Research* **70**(4) 1996–2012.
- Liu, Fei, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, Qingfu Zhang. 2024. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. *Proceedings of the 41st International Conference on Machine Learning*, vol. 235. PMLR, 32201–32223.
- Liu, Mo, Yumo Bai, Meng Qi, Zuo-Jun (Max) Shen. 2026. Inventory management with transformer: Automated decision making for order timing and quantity. *Service Science* **0**(0).
- McKinsey & Company. 2025. Beyond automation: How Gen AI is reshaping supply chains. McKinsey Operations Insights.
- Nadarajah, Selvaprabu, Andre A. Cire. 2025. Self-adapting network relaxations for weakly coupled markov decision processes. *Management Science* **71**(2) 1779–1802.
- Novikov, Alexander, Ng  n V  , Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, Matej Balog. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery.
- Pakiman, Parshan, Selvaprabu Nadarajah, Negar Soheili, Qihang Lin. 2025. Self-guided approximate linear programs: Randomized multi-shot approximation of discounted cost markov decision processes. *Management Science* **71**(4) 3384–3404.
- Porteus, Evan L. 2002. *Foundations of Stochastic Inventory Theory*. Stanford University Press, Stanford, CA.
- Powell, Warren B. 2011. *Approximate Dynamic Programming: Solving the Curses of Dimensionality*. 2nd ed. John Wiley & Sons.
- Romera-Paredes, Bernardino, Mohammadamin Barekatin, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, Alhussein Fawzi. 2024. Mathematical discoveries from program search with large language models. *Nature* **625**(7995) 468–475.
- Scarf, Herbert. 1960. The optimality of (s, s) policies in the dynamic inventory problem. Kenneth J. Arrow, Samuel Karlin, Patrick Suppes, eds., *Mathematical Methods in the Social Sciences*. Stanford University Press, Stanford, CA, 196–202.
- van Jaarsveld, Willem, Joachim Arts. 2024. Projected inventory-level policies for lost sales inventory systems: Asymptotic optimality in two regimes. *Operations Research* **72**(5) 1790–1805.
- Veinott, Arthur F., Jr., Harvey M. Wagner. 1965. Computing optimal (s, S) inventory policies. *Management Science* **11**(5) 525–552.
- Xin, Linwei. 2021. Understanding the performance of capped base-stock policies in lost-sales inventory models. *Operations Research* **69**(1) 61–70.

- Xin, Linwei. 2026. Capped base-stock policies: A 2.33-approximation. *Working Paper* **0**(0).
- Ye, Haoran, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, Guojie Song. 2024. Reevo: Large language models as hyper-heuristics with reflective evolution. *Advances in Neural Information Processing Systems*, vol. 37.
- Zhang, Kairen, Xiangyu Gao, Zhanyue Wang, Sean X Zhou. 2025. Sampling-based approximation for series inventory systems. *Management Science* **71**(10) 8200–8217.
- Zipkin, Paul. 2008. Old and new methods for lost-sales inventory systems. *Operations Research* **56**(5) 1256–1263.
- Zipkin, Paul H. 2000. *Foundations of Inventory Management*. McGraw-Hill, Boston, MA.

This page is intentionally blank. Proper e-companion title page, with INFORMS branding and exact metadata of the main paper, will be produced by the INFORMS office when the issue is being assembled.

EC.1. Prompt Template

We report the prompt template used in the LLM-guided policy-generation step of AIPS. The prompt is reproduced verbatim. Placeholders represented in double braces indicate implementation-time substitutions; in the experiments, they are replaced by instance-specific inputs, including the training demand trajectories, selected parent-policy code, optimized parameter values, and empirical cost statistics. When the parent set contains multiple policies, the parent-policy block is repeated with the corresponding code and statistics.

Time is indexed differently in the prompt than in the main paper. In the main paper, the T -period cost-incurring horizon is indexed by $t = 1, \dots, T$. In the prompt, we prepend L zero-demand, zero-cost periods, so that each simulated trajectory spans $L + T$ periods. We refer to the prepended periods $t = 1, \dots, L$ as the “planning phase” and the remaining periods $t = L + 1, \dots, L + T$ as the “selling phase.” The selling phase corresponds one-to-one to the T periods in the main paper. The planning phase is used solely to initialize the pipeline: orders placed during these L periods become arrivals during the first L periods of the selling horizon. Thus, the expanded indexing in the prompt makes the pre-horizon pipeline initialization explicit.

[BEGIN PROMPT]

Section 1. Problem Statement

We consider an inventory control problem for a single product over a finite discrete horizon. The system has a selling horizon of T periods and a fixed, deterministic delivery lead time L . An order placed at the beginning of period t arrives at the beginning of period $t+L$.

There are two phases:

1. Planning Phase ($t = 1, \dots, L$):
No customer demand occurs. The manager may place orders, but no costs are incurred. In the historical data, these periods appear as zeros, which are placeholders indicating absence of demand.
2. Selling Phase ($t = L+1, \dots, L+T$):
Customer demand occurs and costs are incurred. The objective is to minimize total cost over these T selling periods.

At the beginning of each period $t = 1, \dots, L+T$, the manager observes:

- On-hand inventory $I_t \geq 0$.
- Pipeline vector $Q_t = (q_{t,1}, q_{t,2}, \dots, q_{t,L})$,
where $q_{t,k}$ is the quantity scheduled to arrive at the beginning of period $t+k-1$.

We assume $I_1 = 0$.

At the start of period t , the order $q_{t,1}$ arrives. Before demand, the available inventory is $I_t + q_{t,1}$. The manager then places an order $a_t \geq 0$, which will arrive at period $t+L$.

After ordering, demand D_t is realized. Sales and lost sales are:

```

S_t = min(I_t + q_{t,1}, D_t),
lost_sales_t = max(0, D_t - I_t - q_{t,1}).

```

The next on-hand inventory is:

```

I_{t+1} = max(0, I_t + q_{t,1} - D_t).

```

Pipeline orders shift forward and the new order enters the last slot:

```

Q_{t+1} = (q_{t,2}, q_{t,3}, ..., q_{t,L}, a_t).

```

During the selling phase ($t = L+1, \dots, L+T$), the manager incurs:

- holding cost $h * \max(0, I_t + q_{t,1} - D_t)$,
- lost-sales cost $p * \max(0, D_t - I_t - q_{t,1})$.

Thus the period- t cost is:

```

c(I_t, q_{t,1}, D_t)
= h * max(0, I_t + q_{t,1} - D_t)
+ p * max(0, D_t - I_t - q_{t,1}).

```

A stationary policy π maps each state (I_t, Q_t) to an order quantity:

```

a_t = \pi(I_t, Q_t).

```

We are given N historical demand trajectories:

```

D^n = (D_1^n, ..., D_L^n, D_{L+1}^n, ..., D_{L+T}^n),
n = 1, ..., N,

```

where $D_1^n, \dots, D_L^n$ are zeros indicating no demand in the planning phase, and the remaining T entries are actual demands.

The objective is to find a stationary policy minimizing the average total cost:

```

min_{\pi} (1/N) * \sum_{n=1}^N \sum_{t=L+1}^{L+T}
c(I_t^{\pi,n}, q_{t,1}^{\pi,n}, D_t^n).

```

The simulator evaluates a policy using the following procedure:

For each demand trajectory $n = 1, \dots, N$:

```

Set I_1 = 0
Set Q_1 = (0, 0, ..., 0) # length L

```

For each period $t = 1, \dots, L+T$:

```

q_{t,1} = first element of Q_t
a_t = policy(I_t, Q_t)
D_t = D_t^n

```

If $t > L$:

```

Add c(I_t, q_{t,1}, D_t) to total cost

```

```

I_{t+1} = max(0, I_t + q_{t,1} - D_t)
Q_{t+1} = (q_{t,2}, q_{t,3}, ..., q_{t,L}, a_t)

```

Return the average total cost over the historical demand trajectories.

Section 2. Problem Parameters and Training Data

Problem parameters:

- Selling phase horizon: $T = 50$ periods
- Lead time: $L = 6$ periods
- Holding cost: $h = 1$ per unit per period
- Lost-sales cost: $p = 2$ per unit

Historical demand trajectories:
`{{TRAINING_DEMAND_TRAJECTORIES}}`

Each trajectory has length $L+T$. The first L entries are zeros corresponding to the planning phase, and the final T entries are selling-phase demands.

Section 3. Main Task

Given the problem parameters and historical demand trajectories above, use the historical parent policy and its corresponding cost statistics in Section 4 to generate a modified version of the policy. The modified policy should aim to achieve a lower average total cost on the historical demand trajectories.

Each policy is represented by Python code. The function must be named:

```
compute_order_amount
```

The function accepts two inputs:

- `on_hand_inventory` (float): on-hand inventory I_t at the start of the period.
- `pipeline_orders` (list[float]): pipeline orders
 $Q_t = (q_{\{t,1\}}, q_{\{t,2\}}, \dots, q_{\{t,L\}})$, indexed from oldest to newest.

The function returns one output:

- `order_amount` (int): order quantity a_t for this period, with $order_amount \geq 0$.

The list `pipeline_orders` is a FIFO queue of length L :

- `pipeline_orders[0]` is $q_{\{t,1\}}$, the oldest pipeline order arriving at the beginning of the current period.
- `pipeline_orders[-1]` is $q_{\{t,L\}}$, the newest pipeline order.

Instructions for Generating the Modified Policy

Follow these requirements when generating the modified policy:

1. Mark optimizable parameters in the code using comments of the form:
`# OPT_PARAM: {"initial": value, "min": lower, "max": upper, "type": "float"}`

Example:

```
base_stock = 500.0 # OPT_PARAM: {"initial": 500.0, "min": 300.0, "max": 800.0, "type": "float"}
```

2. The `OPT_PARAM` comment must appear on the same line as the parameter assignment.
3. Only mark parameters that are assigned within the function body.
4. Only mark parameters assigned with an equals sign.
5. Do not mark more than 10 optimizable parameters.
6. The policy must be strictly stationary:
 - The order amount may depend only on `on_hand_inventory` and `pipeline_orders`.
 - The order amount may not depend on time, step counters, episode index, demand trajectory index, or previous calls.
 - Do not use hidden memory, function attributes, cached state, global variables, module-level variables, or closures that store information across calls.
7. The function must always return a finite nonnegative integer order quantity.

8. After generating the modified implementation, provide a concise explanation for the changes inside double curly braces, like this:
 {{Your explanation here.}}
9. Do not provide any additional explanations outside the code and the double curly-brace explanation.

Section 4. Historical Parent Policy and Cost Statistics

Parent policy 1:

```
def compute_order_amount(on_hand_inventory, pipeline_orders):
    base_stock = 497.94 # OPT_PARAM: {"initial": 497.94, "min": 400, "max": 700, "type": "float"}
    safety_stock = 25.03 # OPT_PARAM: {"initial": 25.03, "min": 20, "max": 100, "type": "float"}
    demand_forecast = 86.14 # OPT_PARAM: {"initial": 86.14, "min": 80, "max": 110, "type": "float"}
    smoothing_factor = 0.05 # OPT_PARAM: {"initial": 0.05, "min": 0.05, "max": 0.30, "type": "float"}
    pipeline_weight = 0.50 # OPT_PARAM: {"initial": 0.50, "min": 0.50, "max": 1.00, "type": "float"}

    weighted_pipeline = 0.0
    for i, q in enumerate(pipeline_orders):
        weight = pipeline_weight ** (len(pipeline_orders) - i - 1)
        weighted_pipeline += q * weight

    net_inventory = on_hand_inventory + weighted_pipeline

    pipeline_sum = sum(pipeline_orders)
    if pipeline_sum > 0:
        pipeline_ratio = weighted_pipeline / pipeline_sum
        adjusted_safety = safety_stock * (1.0 + 0.5 * (1.0 - pipeline_ratio))
    else:
        adjusted_safety = safety_stock

    target_inventory = base_stock + adjusted_safety
    order_up_to = max(0.0, target_inventory - net_inventory)

    smoothed_order = (
        smoothing_factor * order_up_to
        + (1.0 - smoothing_factor) * demand_forecast
    )

    order_amount = max(0, int(round(smoothed_order)))
    return order_amount
```

Cost statistics of parent policy 1 on the historical demand trajectories:

- Average total cost per period: 14.76
- Standard deviation of total cost per period: 14.14
- Average holding cost per period: 5.67
- Average lost-sales cost per period: 9.10
- Holding-to-lost-sales ratio: 5.7 : 9.1
- Per-trajectory total cost:
 - mean = 826.80
 - standard deviation = 163.23
 - range = (494.00, 1584.00)

Now generate one modified policy.

[END PROMPT]

EC.2. Additional Results for Section 4

Table EC.1 reports the mean and standard error of the cost reduction percentage across different problem instances and generations. Each problem instance occupies two rows: the first row reports the mean of $\text{CR}_{k,r}^{(g)}$ over the ten repeats, and the parenthesized second row reports its standard error. In the instance names, **Exp** denotes Exponential demand; **L2**, **L4**, and **L6** denote the lead time; **CR2** and **CR5** denote the cost ratio $p/h = 2$ and $p/h = 5$; and **Normal10**, **Normal30**, and **Normal50** denote Normal demand with $\sigma = 10, 30, 50$, respectively.

The mean and standard error of cost reduction for a subset of problem instances $\mathcal{K} \subset [K]$ are computed as

$$\text{MCR}^{(g)}(\mathcal{K}) := \frac{1}{|\mathcal{K}|} \sum_{k \in \mathcal{K}} \sum_{r=1}^R \text{CR}_{k,r}^{(g)}, \quad \text{SE}^{(g)}(\mathcal{K}) := \frac{\text{stdev}(\text{CR}_{k,r}^{(g)} : k \in \mathcal{K}, r = 1, \dots, R)}{\sqrt{|\mathcal{K}| \cdot R}}.$$

EC.3. Construction of the Robustness Demand Set

This appendix describes the parameter grids and simulation procedures used to construct the demand specifications in the cross-environment robustness analysis. We consider 13 parametric demand families, covering bounded and unbounded distributions, continuous and discrete distributions, and demand environments with different degrees of variability, skewness, and tail behavior.

EC.3.1. Trajectory Generation

A demand specification is defined by a distribution family $\mathcal{F}$ and a vector of family-specific parameters. For each specification, we generate integer-valued demand trajectories of length $T = 50$. Demand observations from discrete distributions are sampled directly from their integer-valued support. For a continuous distribution, we first generate a continuous random variable $\tilde{D}_t$ and then construct integer demand according to

$$D_t = \max \left\{ 0, \text{round}(\tilde{D}_t) \right\}.$$

Thus, continuous draws are rounded to the nearest integer, and any negative rounded values are replaced by zero.

For each demand specification, we generate 100 training trajectories and 1,000 separate test trajectories. The training trajectories are used to optimize the parameters of each policy class within the corresponding environment, whereas the test trajectories are reserved for out-of-sample performance evaluation.

Table EC.1 Mean (standard error) of $CR_{k,r}^{(g)}$ by problem instance and generation

Instance	$g = 1$	$g = 2$	$g = 3$	$g = 4$	$g = 5$	$g = 6$	$g = 7$	$g = 8$	$g = 9$	$g = 10$
Poisson_L2_CR2	10.98 (3.51)	16.68 (3.19)	20.09 (2.68)	22.74 (2.54)	25.02 (2.53)	25.79 (2.64)	26.19 (2.69)	28.41 (1.08)	28.50 (1.10)	28.53 (1.11)
Poisson_L2_CR5	7.52 (1.85)	11.58 (1.10)	12.53 (1.27)	14.71 (1.50)	16.34 (1.28)	16.91 (1.28)	17.36 (1.13)	17.62 (1.08)	17.87 (1.12)	17.91 (1.12)
Poisson_L4_CR2	24.37 (6.47)	41.15 (4.84)	47.30 (3.58)	48.88 (3.58)	52.86 (1.64)	54.01 (1.27)	55.02 (0.80)	55.38 (0.72)	55.86 (0.67)	56.16 (0.60)
Poisson_L4_CR5	17.95 (5.59)	31.67 (3.11)	36.82 (2.76)	40.28 (2.13)	42.00 (2.10)	43.55 (1.67)	44.69 (1.06)	45.55 (0.73)	45.68 (0.79)	45.88 (0.79)
Poisson_L6_CR2	54.26 (8.40)	56.99 (7.40)	60.74 (6.31)	62.65 (6.27)	69.46 (2.88)	70.41 (1.94)	70.95 (1.42)	71.02 (1.35)	71.57 (0.82)	71.58 (0.82)
Poisson_L6_CR5	44.99 (8.60)	55.24 (6.54)	57.51 (6.23)	60.58 (3.92)	63.61 (1.18)	64.75 (0.32)	64.85 (0.31)	64.87 (0.29)	64.87 (0.29)	64.93 (0.30)
Normal10_L2_CR2	11.00 (3.85)	18.74 (4.01)	24.72 (2.86)	25.25 (2.93)	25.47 (2.91)	25.80 (2.90)	25.97 (2.93)	28.85 (1.10)	28.99 (1.11)	29.03 (1.12)
Normal10_L2_CR5	5.12 (2.12)	12.35 (2.18)	15.49 (1.95)	16.66 (1.71)	17.05 (1.67)	17.52 (1.68)	17.55 (1.68)	18.42 (1.82)	18.44 (1.81)	18.51 (1.84)
Normal10_L4_CR2	45.64 (3.43)	52.05 (1.74)	52.84 (1.63)	53.08 (1.57)	53.25 (1.56)	53.38 (1.58)	54.33 (0.99)	55.05 (0.61)	55.07 (0.62)	55.18 (0.58)
Normal10_L4_CR5	6.02 (3.98)	12.98 (4.93)	18.36 (5.66)	22.54 (5.75)	27.41 (5.08)	34.08 (4.44)	34.47 (4.49)	35.02 (4.40)	35.21 (4.37)	35.36 (4.27)
Normal10_L6_CR2	47.54 (8.78)	64.66 (4.22)	68.00 (2.52)	70.74 (1.83)	70.74 (1.83)	71.92 (0.88)	72.32 (0.78)	72.32 (0.78)	72.65 (0.57)	72.70 (0.55)
Normal10_L6_CR5	34.13 (8.94)	46.71 (6.93)	49.98 (6.28)	57.01 (2.82)	58.88 (2.39)	59.69 (2.41)	59.71 (2.41)	60.68 (1.46)	60.96 (1.22)	61.94 (0.43)
Normal30_L2_CR2	5.28 (1.71)	8.28 (1.25)	11.00 (0.42)	11.29 (0.43)	11.33 (0.43)	11.51 (0.45)	11.59 (0.47)	11.74 (0.45)	11.81 (0.42)	11.85 (0.42)
Normal30_L2_CR5	3.71 (1.28)	4.96 (1.07)	6.58 (0.99)	6.70 (0.99)	7.24 (1.02)	7.75 (0.66)	7.95 (0.66)	8.30 (0.54)	8.50 (0.53)	8.68 (0.56)
Normal30_L4_CR2	20.98 (4.58)	29.21 (1.90)	31.77 (0.81)	32.51 (0.46)	32.61 (0.48)	33.08 (0.32)	33.21 (0.27)	33.32 (0.27)	33.50 (0.24)	33.52 (0.23)
Normal30_L4_CR5	18.61 (2.72)	20.62 (1.72)	22.24 (1.41)	23.88 (0.64)	24.17 (0.45)	24.44 (0.29)	24.63 (0.22)	24.74 (0.15)	24.80 (0.16)	24.86 (0.15)
Normal30_L6_CR2	29.88 (4.57)	33.98 (3.05)	36.51 (2.84)	38.00 (2.88)	40.52 (2.54)	40.77 (2.54)	41.67 (2.19)	44.54 (0.53)	44.82 (0.52)	44.83 (0.52)
Normal30_L6_CR5	12.66 (4.15)	25.48 (3.87)	28.92 (2.64)	31.65 (1.57)	32.94 (1.01)	33.08 (1.03)	33.55 (0.97)	34.03 (0.72)	34.22 (0.75)	34.45 (0.66)
Normal50_L2_CR2	6.77 (1.54)	8.54 (1.20)	9.40 (1.22)	9.72 (1.13)	11.53 (0.22)	11.72 (0.19)	11.84 (0.16)	11.90 (0.18)	11.97 (0.16)	12.03 (0.12)
Normal50_L2_CR5	3.50 (0.88)	4.55 (0.44)	4.97 (0.37)	5.31 (0.36)	5.87 (0.37)	5.92 (0.37)	5.92 (0.37)	6.06 (0.37)	6.08 (0.37)	6.22 (0.33)
Normal50_L4_CR2	11.00 (2.39)	16.75 (0.54)	17.23 (0.58)	17.42 (0.60)	17.83 (0.52)	17.93 (0.51)	18.05 (0.50)	18.54 (0.40)	18.95 (0.38)	19.01 (0.39)
Normal50_L4_CR5	9.89 (1.74)	10.50 (1.69)	10.96 (1.77)	11.81 (1.63)	11.88 (1.64)	13.54 (0.66)	13.72 (0.69)	13.86 (0.69)	14.05 (0.71)	14.05 (0.71)
Normal50_L6_CR2	20.11 (3.75)	24.41 (3.50)	28.31 (1.56)	28.80 (1.13)	29.29 (0.76)	29.75 (0.65)	29.79 (0.66)	30.39 (0.33)	30.49 (0.27)	30.83 (0.19)
Normal50_L6_CR5	14.75 (2.92)	17.72 (2.33)	18.31 (2.35)	20.21 (2.24)	21.80 (1.43)	22.54 (1.43)	24.16 (0.43)	24.59 (0.27)	24.73 (0.19)	24.86 (0.20)
Exp_L2_CR2	5.77 (0.84)	7.30 (0.59)	8.36 (0.13)	8.47 (0.12)	8.54 (0.12)	8.60 (0.11)	8.63 (0.12)	8.66 (0.11)	8.76 (0.10)	8.83 (0.11)
Exp_L2_CR5	6.54 (0.81)	7.79 (0.59)	8.16 (0.58)	8.19 (0.57)	8.35 (0.56)	8.94 (0.06)	9.00 (0.03)	9.00 (0.03)	9.01 (0.03)	9.01 (0.03)
Exp_L4_CR2	10.12 (1.14)	11.23 (0.23)	11.33 (0.24)	11.50 (0.25)	11.59 (0.25)	11.60 (0.25)	11.73 (0.16)	11.84 (0.14)	11.90 (0.16)	12.08 (0.08)
Exp_L4_CR5	9.42 (0.72)	9.51 (0.72)	9.87 (0.65)	10.51 (0.24)	10.58 (0.25)	10.63 (0.27)	10.70 (0.27)	10.76 (0.25)	10.83 (0.26)	10.83 (0.26)
Exp_L6_CR2	17.13 (0.74)	17.89 (0.39)	18.06 (0.38)	18.27 (0.36)	18.33 (0.34)	18.35 (0.33)	18.55 (0.26)	18.73 (0.16)	18.89 (0.12)	18.91 (0.12)
Exp_L6_CR5	10.02 (1.57)	13.49 (1.03)	14.51 (0.52)	14.94 (0.49)	14.97 (0.49)	15.15 (0.51)	15.35 (0.44)	15.41 (0.46)	15.67 (0.33)	15.73 (0.34)

Each demand specification is evaluated under four lead times, $L \in \{2, 4, 6, 8\}$, and four lost-sales penalties, $p \in \{2, 4, 6, 10\}$, with the unit holding cost fixed at $h = 1$. Consequently, each demand specification generates $4 \times 4 = 16$ distinct inventory environments. The construction described below produces 629 distinct demand specifications. It therefore yields $629 \times 4 \times 4 = 10,064$ distinct inventory test environments.

EC.3.2. Distribution Families and Parameter Grids

Table EC.2 reports the complete parameter grid used for each demand family. To avoid confusion with the lost-sales penalty, q denotes a probability parameter in the negative-binomial, binomial, and geometric distributions. For the zero-based geometric distribution, the random variable records the number of failures before the first success and therefore has support $\{0, 1, 2, \dots\}$. The negative-binomial distribution is similarly parameterized by the number of failures before r successes.

Table EC.2 Parameter grids used to construct the robustness demand set.

Demand family	Parameter grid and restrictions	Count
<i>(i) Bounded continuous distributions</i>		
Scaled beta	$\tilde{D} = MX$, where $X \sim \text{Beta}(\alpha, \beta)$; $\alpha, \beta \in \{0.5, 1, 2, 5, 10\}$; and $M \in \{20, 50, 100, 200, 500\}$. The raw grid contains 125 combinations; two duplicate moment specifications are removed.	123
Continuous uniform	$\tilde{D} \sim \text{Uniform}(a, b)$, where $a \in \{0, 10, 20, 50\}$ and $b \in \{50, 100, 200, 400, 800\}$, retaining only combinations satisfying $b > a$.	19
Triangular	$\tilde{D} \sim \text{Triangular}(a, c, b)$, where $a \in \{0, 5, 10\}$, $c \in \{20, 50, 100, 200\}$, and $b \in \{60, 120, 250, 500, 800\}$, retaining only combinations satisfying $a < c < b$.	51
<i>(ii) Bounded discrete distributions</i>		
Binomial	$D \sim \text{Binomial}(n, q)$, where $n \in \{20, 50, 100, 200, 400, 800\}$ and $q \in \{0.05, 0.1, 0.2, 0.3, 0.5, 0.7, 0.9, 0.95\}$.	48
Discrete uniform	D is uniformly distributed on $\{\ell, \ell + 1, \dots, u\}$, where $\ell \in \{0, 5, 10, 20, 50\}$ and $u \in \{20, 50, 100, 200, 400, 800\}$, retaining only combinations satisfying $u > \ell$.	27
<i>(iii) Light- to moderate-tailed unbounded distributions</i>		
Gamma	$\tilde{D} \sim \text{Gamma}(k, \theta)$, where $k \in \{0.8, 1, 2, 5, 10, 20\}$ is the shape parameter and $\theta \in \{2, 5, 10, 20, 40, 80\}$ is the scale parameter.	36
Geometric, zero-based	$D \sim \text{Geometric}_0(q)$, where $q \in \{0.03, 0.05, 0.08, 0.1, 0.15, 0.2, 0.3, 0.4, 0.6, 0.8\}$. The support is $\{0, 1, 2, \dots\}$.	10
Negative binomial	$D \sim \text{NegBin}(r, q)$, where $r \in \{2, 5, 10, 20, 40\}$ and $q \in \{0.1, 0.2, 0.3, 0.4, 0.6, 0.8\}$.	30
Normal	$\tilde{D} \sim \mathcal{N}(\mu, \sigma^2)$, where $\mu \in \{20, 50, 80, 100, 150, 200, 300\}$ and $\sigma = f\mu$, with $f \in \{0.10, 0.15, 0.20, 0.25, 0.30\}$. The specifications $(\mu, \sigma) = (100, 10)$ and $(100, 30)$, already used in the main experiments, are excluded.	33
Weibull	$\tilde{D} \sim \text{Weibull}(k, \lambda)$, where $k \in \{0.8, 1.2, 1.8, 2.5, 3.5, 5.0\}$ is the shape parameter and $\lambda \in \{10, 30, 60, 100, 200, 500\}$ is the scale parameter.	36
<i>(iv) Heavy-tailed or highly skewed distributions</i>		
Lognormal	$\tilde{D} \sim \text{Lognormal}(\mu_{\log}, \sigma_{\log}^2)$, where $\mu_{\log} \in \{1.5, 2.0, 2.5, 3.0, 3.5, 4.0\}$ and $\sigma_{\log} \in \{0.3, 0.5, 0.7, 1.0, 1.3\}$. These are the parameters of the underlying normal distribution.	30
Pareto type I	$\tilde{D} \sim \text{ParetoI}(\alpha, x_m)$, where $\alpha \in \{1.3, 1.5, 2.0, 3.0, 4.0, 6.0\}$ is the shape parameter and $x_m \in \{1, 5, 10, 20, 50, 100\}$ is the minimum support value.	36
Zero-inflated negative binomial	Demand equals zero with probability $\pi_0 \in \{0.1, 0.3, 0.5, 0.7, 0.85\}$. Otherwise, $D \sim \text{NegBin}(r, q)$, where $r \in \{2, 5, 10, 20, 40\}$ and $q \in \{0.1, 0.2, 0.3, 0.4, 0.6, 0.8\}$.	150
Total		629

Table EC.3 Test-set percentage improvement over the optimal base-stock policy at the final generation. Each statistic is computed across 10 independent repeats.

Setting and demand distribution	Min (%)	Max (%)	Avg. (%)	Std. dev. (pp)
Without optimizer – Normal	-4.16	22.72	1.57	7.56
Without optimizer – Exponential	0.66	5.68	1.18	1.58
Without optimizer – Poisson	-1.07	-0.13	-0.79	0.45
With optimizer – Normal	41.58	46.32	45.34	1.57
With optimizer – Exponential	16.13	17.53	17.15	0.40
With optimizer – Poisson	64.78	73.20	72.12	2.65

We retain one representative from each duplicate pair, leaving $125 - 2 = 123$ distinct scaled-beta moment specifications. Together with the specifications from the other 12 distribution families, this produces the final set of 629 demand specifications.

EC.4. Test-Set Performance for the Value of Optimization

This appendix examines the impact of the optimizer on test-set performance for our AIPS algorithm. As mentioned in Section 6 of the main body, the test-set performance is very close to the training-set performance reported in the main body.

In Table EC.3, the results without the external optimizer are evaluated at generation 20, whereas results with the external optimizer are evaluated at generation 10. These results are consistent with the training-set performance reported in the main body, showing that incorporating external optimizers can significantly improve the performance of the proposed AIPS algorithm.

EC.5. LLM Backbone

This section provides additional results for the LLM-backbone analysis in §7. Table EC.4 complements Figure 12 by reporting the generation-by-generation performance of each backbone with and without the external optimizer. The rows correspond to generations 1–20. For each backbone, the two columns report results without and with the optimizer, respectively. Each entry gives the mean cost reduction relative to the optimized base-stock policy, with the corresponding standard error in parentheses. The optimizer-enabled experiments terminate at generation 10, so the corresponding entries are left blank thereafter. The table shows that optimization improves both the level and speed of performance across all six backbones. For example, at generation 10, mean cost reduction with optimization is 41.9% for GPT-5 Nano, 46.1% for GPT-5 Mini, 45.5% for DeepSeek, 33.5% for Grok 4.1 Fast, 23.1% for Gemini 2.5 Lite, and 46.3% for Gemini 3 Flash. Without optimization, even after 20 generations, the corresponding values are 15.9%, 16.0%, 1.6%, -0.5% , -0.5% , and 42.5%, respectively. The reported standard errors also show substantial heterogeneity across problem instances and repeats, particularly for the stronger-performing backbones.

Table EC.4 Cost reduction across LLM backbones with and without numerical parameter optimization.

Generation	GPT-5 Nano		GPT-5 Mini		DeepSeek		Grok 4.1 Fast		Gemini 2.5 Lite		Gemini 3 Flash	
	Without	With	Without	With	Without	With	Without	With	Without	With	Without	With
1	-50.4 (10.2)	15.8 (3.0)	-0.4 (0.8)	27.7 (5.2)	-1.9 (1.3)	34.2 (4.2)	-59.3 (12.3)	4.5 (1.4)	-25.5 (7.3)	2.8 (0.9)	20.1 (4.5)	44.7 (4.5)
2	-14.3 (6.0)	27.7 (3.6)	1.6 (0.7)	40.4 (4.5)	-1.1 (1.0)	36.7 (3.9)	-37.9 (8.8)	10.6 (2.1)	-14.3 (5.8)	6.0 (1.8)	34.6 (4.8)	45.7 (4.2)
3	-2.1 (3.1)	33.8 (3.9)	2.0 (0.8)	43.4 (4.2)	0.2 (0.9)	38.8 (3.9)	-25.4 (6.8)	13.9 (2.4)	-8.2 (5.0)	8.5 (2.3)	37.6 (4.9)	45.8 (4.2)
4	0.3 (3.0)	37.2 (3.8)	4.5 (1.7)	44.4 (4.2)	0.6 (0.9)	40.0 (4.0)	-15.1 (5.0)	16.6 (2.5)	-5.3 (4.0)	11.2 (2.7)	38.4 (4.9)	45.8 (4.2)
5	3.0 (3.1)	38.4 (3.8)	4.7 (1.7)	45.3 (4.2)	0.7 (0.9)	43.2 (4.0)	-8.4 (3.4)	23.1 (3.3)	-4.8 (3.9)	14.1 (3.3)	38.7 (4.9)	45.8 (4.2)
6	5.7 (3.4)	39.9 (3.9)	6.3 (1.9)	45.5 (4.2)	0.7 (0.9)	43.6 (4.1)	-4.7 (2.4)	27.3 (3.4)	-4.7 (3.9)	16.4 (3.3)	39.6 (4.8)	45.8 (4.2)
7	7.5 (3.9)	40.8 (3.9)	7.8 (1.9)	45.6 (4.1)	0.8 (0.9)	44.1 (4.0)	-4.3 (2.4)	29.6 (3.6)	-3.4 (3.2)	19.4 (3.3)	40.3 (4.7)	46.1 (4.1)
8	8.5 (3.5)	41.5 (3.8)	11.2 (2.5)	45.8 (4.1)	0.9 (1.0)	45.1 (4.0)	-3.7 (2.3)	31.3 (3.8)	-2.2 (2.4)	20.7 (3.5)	40.4 (4.7)	46.3 (4.0)
9	9.4 (3.6)	41.9 (3.8)	12.0 (2.5)	45.9 (4.1)	0.9 (1.0)	45.5 (4.0)	-2.9 (2.1)	33.0 (3.8)	-1.7 (2.2)	22.5 (3.6)	40.6 (4.7)	46.3 (4.0)
10	10.1 (3.7)	41.9 (3.8)	13.0 (2.7)	46.1 (4.1)	0.9 (1.0)	45.5 (4.0)	-2.7 (2.1)	33.5 (3.9)	-1.4 (2.1)	23.1 (3.7)	40.9 (4.7)	46.3 (4.0)
11	10.5 (3.7)	—	13.8 (2.9)	—	0.9 (1.0)	—	-2.0 (1.9)	—	-1.0 (2.0)	—	41.2 (4.6)	—
12	10.8 (3.7)	—	14.3 (3.0)	—	0.9 (1.0)	—	-1.6 (1.8)	—	-1.0 (2.0)	—	41.4 (4.6)	—
13	11.6 (3.9)	—	14.7 (3.0)	—	0.9 (1.0)	—	-1.6 (1.8)	—	-0.8 (1.9)	—	41.7 (4.5)	—
14	12.1 (3.9)	—	14.7 (3.0)	—	0.9 (1.0)	—	-1.6 (1.8)	—	-0.7 (1.9)	—	42.0 (4.5)	—
15	12.2 (3.9)	—	15.1 (3.0)	—	0.9 (1.0)	—	-1.6 (1.8)	—	-0.5 (1.7)	—	42.1 (4.5)	—
16	13.2 (4.0)	—	15.1 (3.0)	—	1.4 (0.9)	—	-1.6 (1.8)	—	-0.5 (1.7)	—	42.1 (4.5)	—
17	13.8 (4.0)	—	15.7 (2.9)	—	1.4 (0.9)	—	-1.6 (1.8)	—	-0.5 (1.7)	—	42.1 (4.5)	—
18	15.3 (4.3)	—	15.9 (2.9)	—	1.4 (0.9)	—	-1.6 (1.8)	—	-0.5 (1.7)	—	42.2 (4.5)	—
19	15.8 (4.3)	—	16.0 (2.9)	—	1.4 (0.9)	—	-1.1 (1.7)	—	-0.5 (1.7)	—	42.2 (4.5)	—
20	15.9 (4.3)	—	16.0 (2.9)	—	1.6 (0.8)	—	-0.5 (1.6)	—	-0.5 (1.7)	—	42.5 (4.5)	—

Notes. Entries report the mean cost reduction (in %), rounded to one decimal place, with the standard error shown in parentheses. “Without” and “With” indicate whether numerical parameter optimization is excluded from or included in the policy-search procedure, respectively. Larger values indicate greater cost reduction. A dash indicates that the corresponding optimizer-enabled experiment was not conducted.

Table EC.5 reports the corresponding test-set results at the termination of each search. The columns correspond to the six LLM backbones. The upper block reports the no-optimizer results at generation 20, while the lower block reports the optimizer-enabled results at generation 10. Within each block, the rows report the minimum, maximum, mean, and standard deviation of test-set cost reduction across experimental runs. The test-set results reinforce the in-sample findings. With optimization, mean test-set cost reduction ranges from 21.2% for Gemini 2.5 Lite to 45.6% for Gemini 3 Flash. Without optimization, the corresponding means are substantially lower for most backbones, including 14.8% for GPT-5 Nano, 15.0% for GPT-5 Mini, 0.7% for DeepSeek, −1.5%

Table EC.5 Test-set cost reduction at convergence across LLM backbones.

		GPT-5 Nano	GPT-5 Mini	DeepSeek	Grok 4.1 Fast	Gemini 2.5 Lite	Gemini 3 Flash
Without optimizer Generation 20	Min	-81.4	-1.1	-4.2	-24.7	-49.6	9.2
	Max	63.8	63.7	22.7	23.7	0.7	73.1
	Mean	14.8	15.0	0.7	-1.5	-1.5	41.9
	Std. dev.	23.9	16.6	4.4	8.3	9.1	25.2
With optimizer Generation 10	Min	13.0	12.5	16.1	4.3	-7.2	17.3
	Max	73.2	73.2	73.2	73.2	72.8	73.2
	Mean	41.3	45.4	44.9	32.8	21.2	45.6
	Std. dev.	21.7	23.4	22.9	22.0	21.7	23.2

Notes. Entries summarize the test-set cost reduction (in %) across experimental runs at convergence. The without-optimizer results correspond to generation 20, whereas the with-optimizer results correspond to generation 10. Larger values indicate greater cost reduction.

for Grok 4.1 Fast, and -1.5% for Gemini 2.5 Lite. Gemini 3 Flash remains the main exception, achieving 41.9% even without optimization. Thus, the performance patterns shown in Figures 11 and 12 persist on the test set.