

Multi-Facility Location Mechanism Design with Large Language Models

Nguyen Thach¹, Fei Liu², Houyu Zhou³, Hau Chan¹

¹University of Nebraska-Lincoln, ²University of Zurich & ETH Zurich, ³UNSW Sydney

Correspondence: nate.thach@huskers.unl.edu

Abstract

Can large language models (LLMs) synthesize programs that satisfy complex game-theoretic constraints? We investigate this question by applying LLMs to multi-facility location mechanism design, a canonical problem where a social planner must place facilities to minimize total distance to strategic agents while ensuring *strategyproofness*—the property that no agent can benefit by misreporting their preference. This problem provides a challenging testbed for LLM-driven program synthesis given the vast search space (mechanisms are arbitrary Python functions) and the adversarial constraint (robustness against any misreport by any agent). In response, we propose LLMMech, an evolutionary framework that embeds a pre-trained LLM within a search loop to generate interpretable, empirically strategyproof mechanisms as Python code. Unlike prior neural approaches, LLMMech requires no handcrafted features or hyperparameter tuning. To prevent premature convergence in the constrained search space, we introduce *prompt evolution*—an automatic procedure for evolving the prompts themselves. Experiments on both unweighted and weighted settings show that LLMMech outperforms existing hand-crafted and deep learning-based mechanisms across various problem settings. The resulting mechanisms are also interpretable and generalizable to unseen distributions and instance sizes.

1 Introduction

Large language models (LLMs) have demonstrated remarkable capabilities in understanding and generating human-like text (Brown et al., 2020), reasoning about code (Jiang et al., 2024; Chen et al., 2021), solving complex optimization problems (Liu et al., 2024b), and discovering novel algorithms (Romera-Paredes et al., 2024). A natural question arises: Can LLMs go beyond generating code for well-specified tasks and instead *synthesize mechanisms*—algorithms that must satisfy complex

game-theoretic constraints—for problems where traditional design approaches struggle?

In this paper, we investigate this question in the context of multi-facility location mechanism design, a canonical problem at the intersection of operations research, economics, and AI (Chan et al., 2021; Moulin, 1980; Procaccia and Tennenholtz, 2013). In this problem, a social planner must locate K facilities (e.g., clinics, transit stations) to serve a population of strategic agents, each of whom reports a preferred location. The challenge is to design a *strategyproof* mechanism: One where no agent can benefit by misreporting their preference, regardless of what others report. The mechanism must simultaneously minimize *social cost*—the total (weighted) distance between agents and their nearest facility.

This problem provides an ideal testbed for studying LLM-driven program synthesis under constraints for three reasons. First, the search space is vast: Mechanisms are arbitrary functions mapping agent reports to facility locations, representable as short Python programs. Second, the constraint (strategyproofness) is adversarial in nature—a mechanism must be robust against *any* misreport by *any* agent—making it fundamentally harder than standard optimization. Third, theoretical results establish that no strategyproof mechanism can achieve a bounded approximation ratio for more than two facilities (Fotakis and Tzamos, 2014; Escoffier et al., 2011).

Existing Approaches and Their Limitations.

The mechanism design literature has pursued two main approaches (Chan et al., 2021). The first relies on manual, domain-expert design of mechanisms with theoretical approximation guarantees—a labor-intensive process that often yields mechanisms with poor worst-case performance for more than two facilities. The second approach, introduced by Golowich et al. (2018), uses deep learning

to learn approximately strategyproof mechanisms from location data. While empirically effective, these neural approaches suffer from limited interpretability (the learned mechanisms are opaque neural networks), require careful hyperparameter tuning of network architectures and learning rates, and struggle to generalize to out-of-distribution agent preferences or unseen instance sizes.

Therefore, in this paper, we strive to address the following question.

Can LLMs serve as effective program synthesizers for constrained economic design tasks, producing mechanisms that are hyperparameter-free, interpretable, approximately strategyproof, and optimal empirically?

Our Approach. Inspired by recent advances in LLM-based code generation (Chen et al., 2021; Jiang et al., 2024), algorithm discovery (Romera-Paredes et al., 2024; Liu et al., 2024a), and agentic reasoning (Yao et al., 2023; Shinn et al., 2023), we propose **LLMMech**, the first evolutionary framework that leverages LLMs to automatically synthesize interpretable, empirically strategyproof mechanisms for multi-facility location. Our approach embeds a pre-trained LLM within an evolutionary search loop: The LLM generates candidate mechanisms as Python code with natural-language descriptions, and the framework evaluates them using a fitness function that jointly optimizes social cost and penalizes violations of strategyproofness (Equation 2). Unlike prior LLM-based optimization methods such as FunSearch (Romera-Paredes et al., 2024) and Evolution of Heuristics (Liu et al., 2024a), our framework introduces *prompt evolution*—an automatic procedure for evolving the prompts themselves—to prevent premature convergence in the search for mechanisms that must satisfy the dual objectives of low cost and strategyproofness.

Our main contributions are as follows:

- **LLM as Mechanism Designer.** We demonstrate that LLMs can serve as effective program synthesizers for a class of constrained economic design problems, producing fully interpretable mechanisms (readable Python code with natural-language descriptions) without requiring domain-specific neural architectures, handcrafted features, or extensive hyperparameter tuning.

- **Dual-Level Evolutionary Framework.** We design a fitness function (Equation 2) that jointly optimizes social cost and empirical strategyproofness, and introduce a prompt evolution component (Section 4.4) that automatically diversifies the search when progress stagnates—addressing a key limitation of existing LLM-based search methods in constrained settings.

- **Empirical Superiority and Generalizability.** Across weighted and unweighted settings with four distributions and up to 25 agents, LLM-Mech outperforms state-of-the-art methods—achieving up to +34% improvement over Moulin-Net (Golowich et al., 2018). Critically, mechanisms discovered on one distribution and instance size generalize to out-of-distribution preferences and larger instances without retraining, a property that neural approaches do not exhibit.

To the best of our knowledge, this is the first study to bring LLMs into multi-facility location mechanism design. We demonstrate how LLMs can automate complex economic design tasks—opening a new avenue for data-driven, yet fully explainable, mechanism synthesis.

Outline. In Section 2, we discuss related work. In Section 3, we provide preliminaries for our considered problem. In Section 4, we elaborate the LLMMech framework. In Section 5, we evaluate its efficacy through various experiments. We conclude our work in Section 6. Readers can find extended related work in Appendix A, complete implementation details in Appendices C–E (which include all missing figures and listings), our breakdown of resource consumption in Appendix F, and additional experiments in Appendix G.

2 Related Work

We discuss related work focusing on automated mechanism design (AMD) and LLMs for (i) code generation and program synthesis as well as (ii) optimization and algorithm design. We also review (iii) mechanism design for facility location and (iv) LLMs as reasoning agents in Appendix A.

Automated Mechanism Design. The literature on AMD dates back to the early 2000s, when Conitzer and Sandholm (2002, 2004) formulated mechanism design as an optimization problem, which can be solved by (integer) linear programming or explicit search, and showed that computing

an optimal *deterministic* mechanism is NP-hard. A more recent line of AMD utilizes neural networks to parameterize and learn mechanisms (Dütting et al., 2019; Duan et al., 2023). Golowich et al. (2018) adapted this idea to multi-facility location, yielding so-called MoulinNet and RegretNet that we adopt as baselines. We refer readers to Curry et al. (2026) for a comprehensive survey on AMD.

Closest to us in spirit, Liu et al. (2025) also employ LLMs to obtain interpretable mechanisms. Their framework targets mechanism design *with payments* (e.g., auctions). Methodologically, we differ in enforcing an explicit empirical strategyproofness penalty *inside* the evolutionary fitness function (Equation 1) and in the prompt evolution component that evolves the variation prompts themselves upon stagnation (Section 4.4).

LLMs for Code Generation and Program Synthesis. Our work is closely related to the growing literature on using LLMs as code generators. Early work such as Codex (Chen et al., 2021) and AlphaCode (Li et al., 2022) demonstrated that LLMs can generate functionally correct programs from natural-language specifications, evaluated primarily via passing unit tests. Subsequent work has explored LLMs for more structured code generation tasks, including generating code with formal specifications (Jiang et al., 2024) and iterative code refinement through self-debugging (Shinn et al., 2023). Our work extends this line of research by asking LLMs to generate code that must satisfy a game-theoretic constraint (strategyproofness) that cannot be verified by simple test cases, requiring instead evaluation over adversarial misreports.

LLMs for Optimization and Algorithm Design. Recent work has integrated LLMs into iterative search frameworks for algorithm design (Liu et al., 2024b): FunSearch (Romera-Paredes et al., 2024) uses a multi-island search for function discovery, whereas EoH (Liu et al., 2024a) evolves heuristics via LLM-driven crossover and mutation (Zhang et al., 2024). However, applying these methods to mechanism design is non-trivial, as the search must jointly optimize the problem objective (social cost) and satisfy a game-theoretic constraint (strategyproofness). Naïvely adopting existing frameworks leads to premature convergence in this expanded search space, motivating our penalty-based fitness function (Equation 2) and prompt evolution component (Section 4.4).

3 Preliminaries

In this section, we begin by presenting the general classical settings of Moulin (1980); Procaccia and Tennenholtz (2013) that were considered in the mechanism design learning problem of Golowich et al. (2018). More specially, there are a set of agents $\mathcal{N} = \{1, \dots, n\}$ and a set of locations $\Omega = [0, 1]$ specifying the domain where facilities can be placed. The social planner is interested in the problem of locating K facilities among locations in Ω . Each agent $i \in \mathcal{N}$ has a preference over Ω for where a facility should be located, represented by a cost function $c_i : \Omega \rightarrow \mathbb{R}$ that associates a real value with each location. Let C_i be the set of permissible cost functions. Let $c = (c_1, \dots, c_n)$ denote a profile of costs, and denote the set of all such tuples by $C = \prod_{i=1}^n C_i$. Further, let $c_{-i} = (c_1, \dots, c_{i-1}, c_{i+1}, \dots, c_n)$ denote all cost functions other than c_i , and $C_{-i} = \prod_{j \neq i} C_j$.

Each agent cares about the location of its closest facility. As such, an agent’s cost function induces a cost for an outcome, $o = (o_1, \dots, o_K)$, which specifies a location for each facility. By a slight abuse of notation, we write $c_i(o) = \min_{k \in \{1, \dots, K\}} c_i(o_k)$. In existing studies, the agents have *single-peaked* preferences.

Definition 1 (Single-peaked preferences). *For locations $\Omega \subseteq \mathbb{R}$, a function $c_i : \Omega \rightarrow \mathbb{R}_{\geq 0}$ is single-peaked if and only if there exists a unique point $a \in \Omega$ (the peak of c_i , denoted by $\tau(c_i)$) such that for all $x, y \in \Omega$: if $x > a$ and $y > x$, then $c_i(y) > c_i(x)$, and if $x < a$ and $y < x$, then $c_i(y) > c_i(x)$.*

As one moves away from the peak $\tau(c_i)$ of cost function c_i , the location becomes less preferred. Following Moulin (1980); Procaccia and Tennenholtz (2013); Golowich et al. (2018) and subsequent work, the cost is defined as $c_i(x) = |x - a|$, where $a = \tau(c_i)$ is the peak.

A *mechanism* for facility location $f : C \rightarrow \Omega^K$ takes reports of agent cost functions as inputs and outputs the locations in Ω for the K facilities. Since agent preferences are single-peaked, with a slight abuse of notation, we instead let f take agent peaks as inputs for convenience. We use $f_k(c)$ to denote the location of the k -th facility for input c . A mechanism is *strategyproof* if agent i cannot strictly decrease their cost by misreporting their cost function, whatever the inputs from others c_{-i} , i.e., for all $i \in \mathcal{N}$, all $c_i, c'_i \in C_i$, we have $c_i(f(c'_i, c_{-i})) \geq c_i(f(c))$.

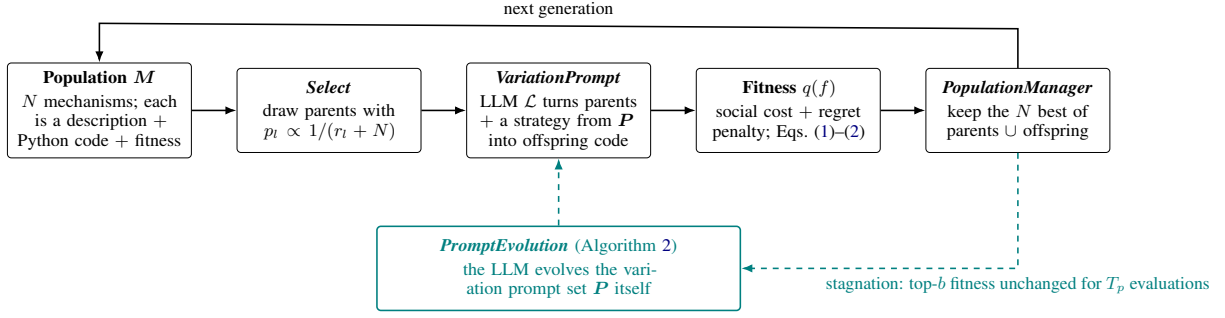


Figure 1: One generation of LLMMech. A population of mechanisms—each a natural-language description paired with a Python function—is evolved by selecting parents, prompting the LLM to write offspring code, scoring every candidate with the fitness of Equations 1–2, and retaining the N best. The prompt evolution component (teal, Algorithm 2), which evolves the variation prompt set P , fires when the fitness of the top- b mechanisms stagnates.

We denote by $\mathcal{M}_{sp} \subseteq \mathcal{M}$ the space of all strategyproof mechanisms for facility location (leaving K implicit). In accordance with Golowich et al. (2018), the agent cost functions are sampled from a joint distribution $\mathcal{D}$ over C , with full support on C . Moreover, $\mathcal{D}$ is a product distribution, i.e., $\mathcal{D} = \prod_{i=1}^n \mathcal{D}_i$ for independent distributions $\mathcal{D}_i$ on C_i .

The goal is to find a strategyproof mechanism $f \in \mathcal{M}_{sp}$ that minimizes the expected (weighted) social cost:

$$g_{sc}(f; \mathcal{D}) = \mathbb{E}_{c \sim \mathcal{D}} \left[\frac{1}{\sum_{i=1}^n \gamma_i} \sum_{i=1}^n \gamma_i c_i(f(c)) \right],$$

where γ_i is the weight associated with agent i and is determined by the social planner who weighs the importance of their preference relative to other agents. We do not have direct access to $\mathcal{D}$, but are provided a sample of profiles $S = \{c^{(1)}, \dots, c^{(R)}\}$ drawn i.i.d. from $\mathcal{D}$. The problem is then to learn the explicit form of f given the profile sample. We define a *problem instance* as an instantiation of the problem, and a *problem setting* as a set of problem instances that share the same configuration of n , K , $\mathcal{D}$, and $\{\gamma_i\}$.

4 LLMs for Mechanism Design

Figure 1 and Algorithm 1 provide an overview of our framework. LLMMech maintains a population of N heuristic mechanisms, denoted as $\mathcal{M} = \{f_1, \dots, f_N\}$. It adopts an evolutionary search framework, iteratively searching for mechanisms that have lower social costs. The mechanisms are designed by prompting pre-trained LLMs. Each mechanism $f_l \in \mathcal{M}$ is evaluated on a set of problem instances and assigned a fitness value $q(f_l)$.

We propose an additional high-level prompt evolution procedure to automatically evolve prompts instead of sticking to hand-crafted prompts to avoid premature convergence in a fixed search pattern. The prompt evolution component is marked in teal, with its main subroutine ‘PromptEvolution’ described in Algorithm 2. The remainder of this section is organized as follows: Section 4.1 describes the components we used in LLMMech to represent a mechanism; Section 4.2 elaborates the abstracted functions in Algorithms 1 and 2; Section 4.3 defines the fitness function for evaluating individual mechanisms in a population; Section 4.4 details the prompt evolution process.

4.1 Mechanism Representation

Each mechanism consists of three parts:

- 1) The *mechanism description* comprises a few sentences in natural language. It is created by LLMs and encapsulates a high-level thought. An example is provided in Figure 4.
- 2) The *code block* implements the mechanism. It should follow a pre-defined format so that it can be identified and seamlessly integrated into the evolutionary framework. In the experiments, we choose to implement it as a Python function¹ (see Appendix B for the code templates used).
- 3) Each mechanism is assigned a *fitness value* to represent its priority in the population, which is used for selection and population management. We elaborate our definition of fitness in Section 4.3.

4.2 Evolutionary Components

We start with the design of prompt strategies. Note that we refer to ‘prompt’ as the entire natural-language input to LLMs, and ‘prompt strategy’

¹Can be written in any programming language.

Algorithm 1 LLMMech

Input: Pre-trained LLM $\mathcal{L}$; Population size N ; Evaluation budget T_e ; Variation prompt set $\mathbf{P}$; **Prompt population size N_p ; Patience $T_p < T_e$; Positive integer $b < N$.**

Output: Mechanism $f^* \in \mathbf{M}$ with best fitness value.

```

1:  $\mathbf{M} \leftarrow \emptyset$ 
2: while  $|\mathbf{M}| < N$  do
3:    $\mathbf{M} \leftarrow \text{InitializationPrompt}()$ 
4: end while
5:  $t \leftarrow |\mathbf{M}|$ 
6:  $\mathbf{M}^* \leftarrow \text{Select}(\mathbf{M}, b)$  %  $|\mathbf{M}^*| = b$ 
7:  $t' \leftarrow 0$ 
8: repeat
9:    $\mathbf{M}' \leftarrow \emptyset$ 
10:  for  $i = 1, 2, \dots, N$  do
11:     $M_i \leftarrow \text{Select}(\mathbf{M})$ 
12:     $M'_i \leftarrow \text{VariationPrompt}(\mathcal{L}, M_i, \mathbf{P})$ 
13:     $\mathbf{M}' \leftarrow \mathbf{M}' \cup M'_i$ 
14:  end for
15:   $\mathbf{M} \leftarrow \text{PopulationManager}(\mathbf{M}, \mathbf{M}', N)$ 
16:   $t \leftarrow t + |\mathbf{M}'|$ 
17:   $\mathbf{M}_{temp}^* \leftarrow \text{Select}(\mathbf{M}, b)$ 
18:  if  $\mathbf{M}_{temp}^* = \mathbf{M}^*$  then
19:     $t' \leftarrow t' + |\mathbf{M}'|$ 
20:  else
21:     $\mathbf{M}^* \leftarrow \mathbf{M}_{temp}^*$ 
22:     $t' \leftarrow 0$ 
23:  end if
24:  if  $t' \geq T_p$  then
25:     $\mathbf{P} \leftarrow \text{PromptEvolution}(\mathcal{L}, \mathbf{P}, N_p)$  % See Algorithm 2
26:  end if
27: until  $t \geq T_e$ 

```

Algorithm 2 PromptEvolution

Input: Pre-trained LLM $\mathcal{L}$; Variation prompt set $\mathbf{P}$; Prompt population size N_p .

Output: Updated $\mathbf{P}$.

```

1:  $\mathbf{P}' \leftarrow \emptyset$ 
2: for  $j = 1, 2, \dots, |\mathbf{P}|$  do
3:    $P_j \leftarrow \text{Select}(\mathbf{P})$ 
4:    $P'_j \leftarrow \text{EvolveVariationPrompt}(\mathcal{L}, P_j)$ 
5:    $\mathbf{P}' \leftarrow \mathbf{P}' \cup P'_j$ 
6: end for
7:  $\mathbf{P} \leftarrow \text{PopulationManager}(\mathbf{P}, \mathbf{P}', N_p)$ 

```

as part of the prompt that provides specific instructions for a certain evolutionary operation (e.g., crossover and mutation). Details of the exact prompts used are illustrated in Appendix C.1–C.2.

Initialization Prompt. In the *Initialization-Prompt* function, we inform the LLM of the facility location of interest and instruct it to design a new mechanism by first presenting the description of the mechanism and the corresponding Python code block (see Figure S7 for an example). We repeat N times to generate N initial mechanisms.

Variation Prompts. We design prompt strategies that either *explore* or *modify* existing mechanisms for generating offsprings. The exploration strate-

gies focus more on the exploration of the space of heuristics by conducting crossover-like operators on parent heuristics. The modification strategies refine a parent heuristic by modifying, adjusting parameters, and removing redundant parts. With a slight abuse of notation, the *VariationPrompt* function denotes the generation of offspring mechanisms by prompting the LLM with parent mechanism(s) M_i and either exploration or modification prompt strategy in $\mathbf{P}$ as input. We only need to provide *one* (instead of two to three) prompt strategy for each type since they are also part of the evolutionary process (to be discussed in Section 4.4). This alleviates the need for extensive prompt engineering, which often requires some degree of domain knowledge especially when dealing with complex problems (Liu et al., 2024b). In our case, we design the modification prompt strategy in accordance to our definition of fitness (to be elaborated in Section 4.3), which only requires basic understandings of the facility location problem, particularly the definition of ‘strategyproofness’. Please refer to Figure S8 (bottom) from Appendix C.2 for exact details.

Selection and Population Management. We rank all mechanisms in the current population according to their fitness. The *Select* function randomly selects one or multiple mechanisms f_l in the current population with probability $p_l \propto 1/(r_l + N)$, where r_l is its rank and N is the population size. The *PopulationManager* function selects the N best mechanisms from the current population (comprising parent mechanisms in $\mathbf{M}$ and offspring mechanisms in $\mathbf{M}'$) to form a population for the next generation.

4.3 Fitness Evaluation

The fitness evaluation process involves running the resulting mechanisms on a problem setting. Following the notations used in Section 3, for a given heuristic mechanism f that takes n agent peaks, $\{\tau(c_1), \dots, \tau(c_n)\}$, as input and outputs the locations of K facilities, $x = (x_1, \dots, x_K)$, we compute its fitness value, $q(f)$, as follows:

$$q(f) = \frac{1}{R \sum_{i=1}^n \gamma_i} \sum_{j=1}^R \sum_{i=1}^n \gamma_i c_i^{(j)}(f(\cdot)) + \theta \left(\max_{i \in \mathcal{N}} \text{rgt}_i(f) \right), \quad (1)$$

where R is the sample size, γ_i is the weight of agent i , and $c_i(x) = \min_{k \in \{1, \dots, K\}} |x_k - \tau(c_i)|$ is the cost of agent i , defined as the distance from the peak of i to its nearest facility. We refer to the first term in Equation 1 as *weighted social cost* under f . We use ‘unweighted’ to refer to the special case of $\gamma_i = 1$ for all agents, and we may omit ‘weighted’ when discussing the general case. The (empirical) regret incurred by f to agent i , rgt_i , is the maximum gain that it can get by misreporting its peaks. Formally,

$$\text{rgt}_i(f) = \frac{1}{R} \sum_{j=1}^R \max_{\ell \in \{1, \dots, B\}} [\max\{0, (\star)\}]$$

$$(\star) = c_i(f(\cdot)) - c_i(f(\cdot)'),$$

where B is the *misreport budget*, i.e., the number of misreports drawn per agent per profile, and $f(\cdot)'$ is the returned facility locations had i misreported its peak. In other words, the input to f is $\{\tau(c_1), \dots, \tau(c'_i), \dots, \tau(c_n)\}$ instead. We refer to $\max_{i \in \mathcal{N}} \text{rgt}_i(f)$ informally as the (empirical) *max regret*, or simply *regret* when the context is clear. Finally, we define θ as a function of max regret incurred by f :

$$\theta(\text{max regret}) = \begin{cases} 1 & \text{if max regret} > \varepsilon \\ 0 & \text{otherwise,} \end{cases} \quad (2)$$

for some (small) value $0 \leq \varepsilon \ll 1$. That is, given a heuristic mechanism f , we penalize its fitness value (social cost initially) if its associated max regret is greater than some predefined threshold. Because social cost is bounded within $[0, 1]$, a mechanism with a fitness value greater than 1 implies that it is not strategyproof.² Positive values of ε relax the strategyproofness constraint.

4.4 Prompt Evolution

During evolution, the diversity in the population is restricted by the manually designed prompt strategies, which may lead to premature convergence at a local optimum. To avoid this, we systematically evolve them whenever stagnation occurs. In particular, when the fitness values of the top- b individuals remain unchanged for T_p consecutive units of evaluation resource (e.g., number of generations), we invoke the prompt evolution process as follows. Throughout the evolutionary framework, we record the fitness value of each prompt strategy, defined

as the average fitness value of the top- d individuals produced by the respective prompt. This helps manage the size of the prompt population as more prompt strategies are generated. When there is stagnation, the prompt population is evolved also via exploration and modification strategies.

Exploration: Given the list of currently used exploration prompts, ask the LLM to design a new exploration prompt that is different from them as much as possible.

Modification: Given the list of currently used modification prompts, ask the LLM to design a new modification prompt that is different from them as much as possible.

In other words, new prompt strategies of each type are introduced after triggering prompt evolution. The *EvolveVariationPrompt* function denotes the generation of offspring prompt strategies by prompting the LLM with parent prompt strategies (of same type) as input. The exact prompt used is shown in Figure S9 from Appendix C.3.

5 Experiments

We start this section by describing the datasets used in Section 5.1 and the considered baselines in Section 5.2. Section 5.3 presents our results for demonstrating the efficacy of LLMMech, followed by a closer look at its learned mechanisms in Section 5.4. We provide an out-of-distribution evaluation in Section 5.5, and conclude with an in-depth analysis of the LLM component in Section 5.6.

5.1 Experimental Setups

The evaluation budget T_e is defined in terms of the maximum number of generations, for which we set to 20 for all problems. The population size is 16, and the maximum running time for each instance is 60 seconds (except when $n = 25$, for which we increase to 180). Appendix E includes all implementation details.

In all problem settings, the dataset for both training and testing phases has 1,000 samples with dimensionality of n , where each entry represents an agent’s peak $\tau(c_i)$. Each sample of (true) peaks is associated with a misreport budget of $B = 10$, i.e., each agent misreports their peak ten times per profile. This follows the empirical strategyproofness protocol of Golowich et al. (2018) (see Appendix G.4 regarding how larger B affects our reported results). The peaks are randomly sampled from uniform, normal ($\mu = 0.5$, $\sigma = 1$), and (two) beta

²The converse does not hold even when $\varepsilon = 0$ for any finite R and B . Hence, ‘strategyproofness’ here is approximate.

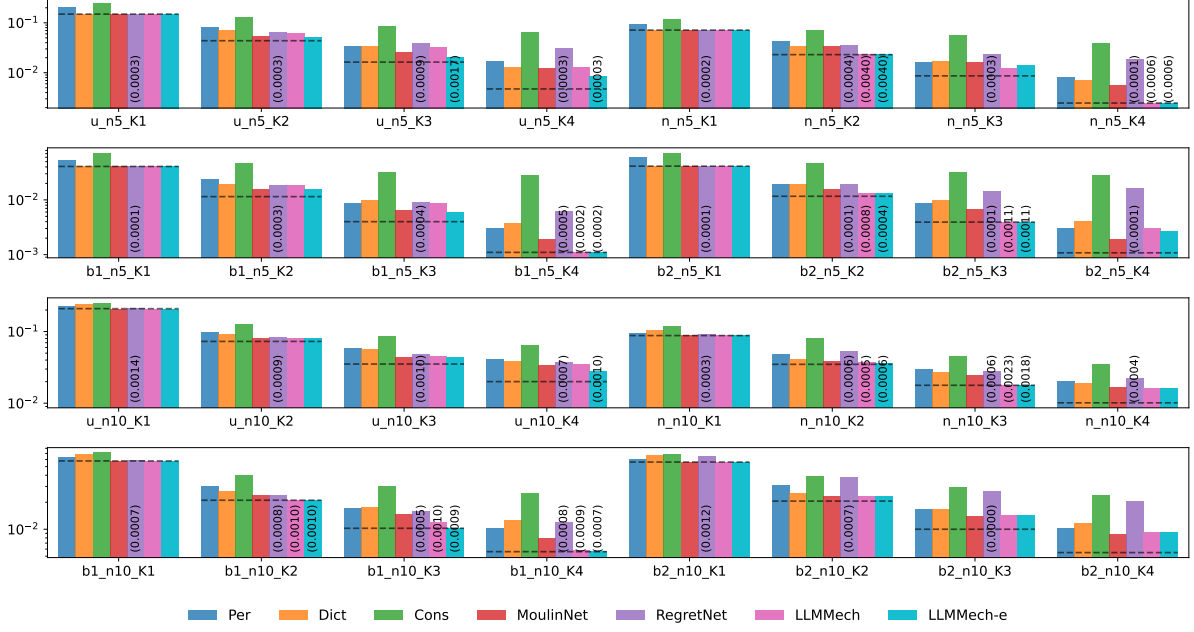


Figure 2: Results for $n = 5$ (top two rows) and $n = 10$ (bottom two rows), weighted social cost. Values on top of bars (if any) represent regret (max across agents) incurred by the mechanism. The dashed lines display values from NonSP. (u—uniform, n—normal, b1—beta1, b2—beta2.)

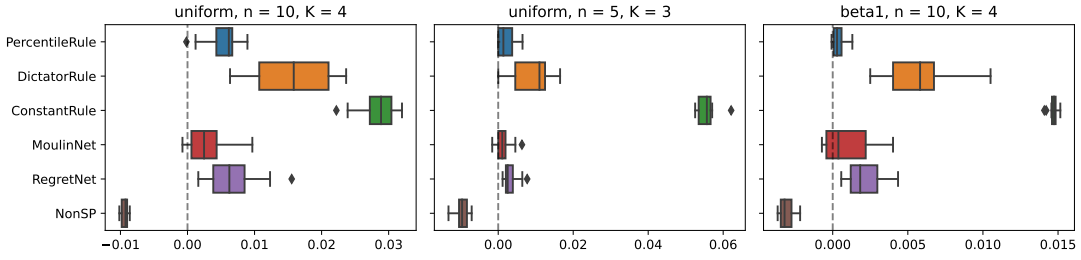


Figure 3: Representative results for arbitrary weights. Each boxplot records net differences in social cost with respect to LLMMech-e.

distributions, referred to as ‘beta1’ ($\alpha = 1, \beta = 9$) and ‘beta2’ ($\alpha = 9, \beta = 1$), defined on $[0, 1]$. For clarity, we refer to our LLM framework as LLM-Mech and the variant with prompt evolution as LLMMech-e.

5.2 Baseline Methods

Our primary goal is to compare LLM-generated mechanisms against the deep learning methods of Golowich et al. (2018) on their own established settings, so we adopt the exact baselines used in their work. (We also consider a classical AMD approach in Appendix G.3 for completeness.) In particular, we consider the neural networks from Golowich et al. (2018), i.e., MoulinNet and RegretNet, as well as the standard mechanisms used in the same work as baselines: the best percentile rule (abbreviated as ‘Per.’) (Sui et al., 2013), the best dictatorial rule (‘Dict.’), and the best constant rule

(‘Cons.’). We also present results for the optimal (non-strategyproof) mechanisms (‘NonSP’). We refer readers to Golowich et al. (2018) for further details of these baselines. We repeat training of both deep learning models with 10 random parameter initializations, and choose the network from the run with smallest regret (based only on the training data). For RegretNet, we fix ρ , the hyperparameter controlling the regularization strength for enforcing strategyproofness (in Equation 6 of the paper), to 1, and the batch size to 50. These values were chosen during our efforts of replicating Table 1 from Golowich et al. (2018).

5.3 Main Results

Weighted Social Cost with Fixed Weights. In this setup, we consider $n = 5, 10$ and adopt the weight configuration used by Golowich et al. (2018) where a few agents (one for $n = 5$ and

two for $n = 10$) are assigned a weight of 5 while others are assigned a weight of 1. Figure 2 shows results on the test set for the (best) mechanisms learned by LLMMech, LLMMech-e, and other baselines. Either of our LLM methods generally yields the best or second-best social cost with negligible or zero regret. We notice whenever $K = 1$, the LLM-generated mechanisms always resemble the generalized median rule for one facility location, which is naturally optimal and strategyproof (Moulin, 1980; Procaccia and Tennenholtz, 2013).

Weighted Social Cost with Arbitrary Weights. We show our framework can effectively deal with arbitrary weight configurations. In this most general setup, we randomly assign agent weights in $\{1, \dots, 5\}$ for 10 different seeds. Table S1 shows the set of weights used for $n = 5$ and $n = 10$. Note that we run both LLMMech and LLMMech-e with ϵ set to 0. In other words, the empirical max regret associated with the reported values is 0.

Figure 3 shows representative results for different combinations of n and K (complete results in Figure S10), where each boxplot records the net differences in social cost with respect to LLMMech-e (which outperforms vanilla LLMMech) over 10 weight sets. We see that even with strict enforcement on strategyproofness, LLMMech-e still generally outperforms other baselines.

Unweighted Social Cost. For completeness, we demonstrate the similarly strong performance of LLM methods on unweighted settings. Figure S11 shows results for $n = 5, 10, 25$, where LLMMech-e seems to outperform the best percentile rule while MoulinNet and RegretNet could not.

5.4 Interpretability of Learned Mechanisms

Our results show that LLMMech and especially LLMMech-e are capable of generating novel, empirically strategyproof mechanisms that outperform existing methods. We now study one of them in detail (Figure 4, with associated code in Appendix D).

For ease of exposition, we have agents reporting peaks $\tau_1, \dots, \tau_n \in [0, 1]$, written in sorted order as $\tau_{(1)} \leq \dots \leq \tau_{(n)}$. The integer weights $\gamma_1, \dots, \gamma_n$ are public constants of the problem instance and are *not* reported by the agents. Let $\Gamma = \sum_{i=1}^n \gamma_i$ be the total weight. A mechanism outputs K facilities $x = (x_1, \dots, x_K)$, agent i 's cost is $c_i(x) = \min_k |x_k - \tau_i|$, and the weighted social cost is $SC_\gamma(\tau, x) = \sum_i \gamma_i c_i(x) / \Gamma$.

Mechanism Description (Generated via LLMMech-e)

The new strategy involves using the K-means clustering algorithm to form clusters from the location samples and selecting the centroid of each cluster as the facility locations to balance the distribution of utility and minimize the total cost.

Figure 4: High-level description of a learned mechanism that yields the lowest cost compared to the baselines and zero empirical regret ('n_n10_K4' in Figure 2).

It can be seen that both the mechanism discovered by LLMMech-e and the baseline percentile rule place facilities at fixed percentile positions, but they differ as follows: The baseline takes unweighted percentiles of the n raw reports and consults the weights only offline, whereas the discovered rule takes weighted percentiles at run time.

In particular, following Golowich et al. (2018), for a fixed position vector $\pi_1 < \dots < \pi_K$ in $\{1, \dots, n\}$, the percentile rule is $f_\pi(\tau) = (\tau_{(\pi_1)}, \dots, \tau_{(\pi_K)})$, i.e., facilities at fixed order statistics of the n raw reports. Training selects the vector π^* that minimizes the average of $SC_\gamma(\tau, f_\pi(\tau))$ over the training profiles.

On the other hand, let $z_{(1)} \leq \dots \leq z_{(\Gamma)}$ be the sorted *expanded* profile in which each report τ_i is repeated γ_i times. With block size $s = \lceil \Gamma / K \rceil$, the mechanism of Figure 4 outputs $x_{k+1} = z_{(ks + \lfloor s/2 \rfloor + 1)}$, $k = 0, \dots, K - 1$, i.e., one facility inside each consecutive block of s expanded points. For this setting ($n = 10$, $K = 4$, $\gamma = (5, 5, 1, \dots, 1)$, hence $\Gamma = 18$ and $s = 5$), the selected positions are 3, 8, 13, 18, so the facilities sit at the weighted quantile levels 3/18, 8/18, 13/18, 18/18 of the reports, rather than at unweighted quantiles of the raw reports.

Strategyproofness Analysis. Similar to the unweighted case, a misreport by agent i moves all γ_i copies of τ_i together. Assuming the agent deviates upward to $\tau'_i > \tau_i$ (the downward case is symmetric), then every order statistic of the expanded profile either stays put or moves upward, and any that moves does so within $[\tau_i, \tau'_i]$. Hence, no facility can end up closer to agent i 's true peak τ_i than it was under truthful reporting, so the agent cannot gain. Because the weights are public constants and cannot themselves be misreported, this argument holds for every profile and every instance, not merely for the sampled misreports. Thus, strategyproofness is exact, not empirical.

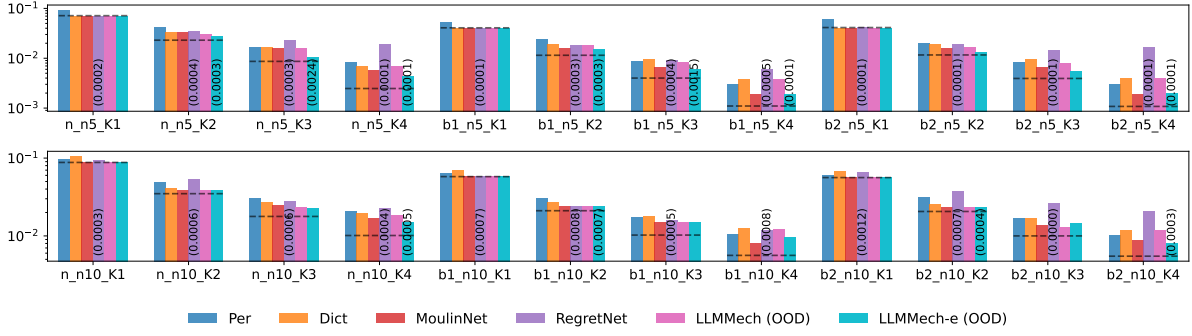


Figure 5: Out-of-distribution (OOD) results for $n = 5$ (top) and $n = 10$ (bottom), weighted social cost. Baselines were trained with in-distribution (uniformly distributed) samples. Values on top of bars (if any) represent regret (max across agents) incurred by the mechanism. The dashed lines display values from NonSP. (n—normal, b1—beta1, b2—beta2.)

Our analysis demonstrates the importance of interpretability that is unattainable for deep learning models. Even when they yield zero empirical regret, we would not be able to verify their strategyproofness on unseen problem instances.

An Example of Learned Prompts. We next scrutinize the prompt strategy behind our considered mechanism. As expected, the prompt strategy, illustrated in Figure 6, was newly introduced during the prompt evolution process. In addition to the overall better performance of LLMMech-e with respect to vanilla LLMMech (Figure 2), this observation further supports the need of an automatic evolution of variation prompts in order to diversify the mechanism population and in turn help the search process escape from local optima.

New Modification Prompt Strategy

Generate a comprehensive analysis of the current parameter settings and their impact on total cost, then propose an alternative strategy with a new set of parameters that aim to balance utility and cost while ensuring that the strategy remains ‘Strategyproof’.

Figure 6: LLM-generated modification prompt strategy that invokes design of the novel mechanism in Figure 4.

5.5 Generalizability of Learned Mechanisms

We investigate the out-of-distribution (OOD) performance of LLM methods. In particular, given the learned mechanisms from LLMMech and LLMMech-e that are trained on uniformly distributed peaks, we test them on instances with peaks generated from other distributions. Figure 5 shows results for $n = 5$ and 10 with fixed weights. In general, LLM methods achieve similar or better

performance with respect to all baselines evaluated on in-distribution peaks. Please refer to Figure S12 for comparison with baselines when they are also evaluated on OOD peaks.

For generalizability of LLM-generated mechanisms to larger instances, please refer to Figure S13 in Appendix G.2.

5.6 LLM Analysis

The preceding experiments demonstrate that LLM-Mech produces effective mechanisms, but leave open the question of *what the LLM contributes* to this outcome. Please refer to Appendix G.5 where we analyze the role of the LLM through ablation studies, a comparison across model families, and an error analysis of failed generations.

6 Conclusion

We propose LLMMech, the first LLM-based evolutionary framework for the automatic design of multi-facility location mechanisms. Moreover, we demonstrate through our various experiments that, with minimal domain knowledge and hyperparameter tuning, LLMMech can generate empirically strategyproof and nearly optimal mechanisms that can generalize to out-of-distribution agent preferences and to larger instances with more agents. By this means, LLMMech can significantly reduce the manual effort involved in traditional mechanism design without compromising interpretability, which is crucial when transparency of the resulting mechanisms is required during social planning.

Limitations

Our work has several limitations that suggest directions for future research.

Empirical vs. Exact Strategyproofness. Our fitness function (Equation 2) enforces strategyproofness only empirically, by penalizing mechanisms whose maximum empirical regret exceeds a threshold ε over a finite sample of misreports. A mechanism that passes this test may still be manipulable on unseen inputs. Given the interpretability of LLM-generated mechanisms, which enables manual verification as demonstrated in Section 5.4, future work could consider a systematic method for certifying the exact strategyproofness of discovered mechanisms in general.

Prompt Sensitivity and Reproducibility. Although LLMMech-e’s prompt evolution component partially mitigates sensitivity to initial prompt design, the stochastic nature of LLM generation means that results vary across runs. Therefore, we carefully report results over three trials for LLMMech and three additional trials for LLMMech-e per LLMMech run.

Applicability Beyond Facility Location. Our end-to-end framework might not be directly applicable to mechanism design problems with payments, e.g., auctions, where the search space is more complex with multiple constraints such as incentive compatibility and individual rationality. Future work could extend our framework to design mechanisms for more general facility location problems, e.g., considering fairness/envy objectives, higher-dimensional space, facilities with constraints, and other types of agent preferences.

Ethical Considerations

Societal Implications of Automated Mechanism Design. Mechanism design for facility location directly informs decisions about the placement of public infrastructure—clinics, transit stations, parks, and other shared resources. Automating this process with LLMs raises the concern that opaque or poorly understood mechanisms could be deployed in high-stakes public planning contexts. While our framework explicitly prioritizes interpretability (the generated mechanisms are readable Python code with natural-language descriptions), we caution that interpretability alone does not guarantee that a mechanism is fair, equitable, or aligned with community values. Social planners should treat LLM-generated mechanisms as candidate solutions requiring domain expert review, not as ready-to-deploy policies.

Potential for Misuse. Our framework could in principle be adapted to design mechanisms that optimize for objectives other than social cost—for instance, mechanisms that favor particular agents or groups. The fitness function (Equation 2) can be modified to encode any objective, including adversarial ones. We recommend that practitioners deploying LLM-generated mechanisms in real-world settings conduct independent audits of the resulting mechanisms for distributional fairness across demographic groups, particularly when agent weights encode socially sensitive attributes.

Environmental and Computational Cost. LLMMech requires between 640 and 1,462 LLM API calls per run (Appendix F), with an estimated cost of \$0.18–\$0.33 using GPT-4o mini. All experiments were conducted on a single machine with a consumer-grade GPU (NVIDIA GeForce RTX 3050 Ti), and we report full resource consumption details in Appendix F to support reproducibility and informed comparison.

Acknowledgments

We thank the anonymous reviewers for the constructive feedback. Nguyen Thach is supported by the National Science Foundation under grant IIS:RI #2414554. Hau Chan is supported by the National Science Foundation under grants IIS:RI #2302999 and IIS:RI #2414554. The content is solely the responsibility of the authors and does not necessarily represent the official views of the funding agencies.

References

- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.
- Hau Chan, Aris Filos-Ratsikas, Bo Li, Minming Li, and Chenhao Wang. 2021. Mechanism design for facility location problems: A survey. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 4356–4365.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Vincent Conitzer and Tuomas Sandholm. 2002. Complexity of mechanism design. In *Proceedings of the*

- Eighteenth Conference on Uncertainty in Artificial Intelligence*, UAI'02, page 103–110, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
- Vincent Conitzer and Tuomas Sandholm. 2004. An algorithm for automatically designing deterministic mechanisms without payments. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '04, page 128–135, USA. IEEE Computer Society.
- Michael J. Curry, Zhou Fan, Yanchen Jiang, Sai Srivatsa Ravindranath, Tonghan Wang, and David C. Parkes. 2026. [Automated mechanism design: A survey](#). *SIGecom Exch.*, 22(2):102–120.
- Zhijian Duan, Haoran Sun, Yurong Chen, and Xiaotie Deng. 2023. A scalable neural network for dsic affine maximizer auction design. *Advances in Neural Information Processing Systems*, 36:56169–56185.
- Paul Dütting, Zhe Feng, Harikrishna Narasimhan, David Parkes, and Sai Srivatsa Ravindranath. 2019. Optimal auctions through deep learning. In *International Conference on Machine Learning*, pages 1706–1715. PMLR.
- Bruno Escoffier, Laurent Gournes, Nguyen Kim Thang, Fanny Pascual, and Olivier Spanjaard. 2011. Strategy-proof mechanisms for facility location games with many facilities. In *Algorithmic Decision Theory: Second International Conference, ADT 2011, Piscataway, NJ, USA, October 26-28, 2011. Proceedings 2*, pages 67–81. Springer.
- Dimitris Fotakis and Christos Tzamos. 2014. On the power of deterministic mechanisms for facility location games. *ACM Transactions on Economics and Computation (TEAC)*, 2(4):1–37.
- Noah Golowich, Harikrishna Narasimhan, and David C Parkes. 2018. Deep learning for multi-facility location mechanism design. In *IJCAI*, pages 261–267.
- Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.
- Fei Liu, Tong Xialiang, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. 2024a. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Forty-first International Conference on Machine Learning*.
- Fei Liu, Yiming Yao, Ping Guo, Zhiyuan Yang, Xi Lin, Xialiang Tong, Mingxuan Yuan, Zhichao Lu, Zhenkun Wang, and Qingfu Zhang. 2024b. A systematic survey on large language models for algorithm design. *arXiv preprint arXiv:2410.14716*.
- Jiayuan Liu, Mingyu Guo, and Vincent Conitzer. 2025. An interpretable automated mechanism design framework with large language models. *arXiv preprint arXiv:2502.12203*.
- Pinyan Lu, Xiaorui Sun, Yajun Wang, and Zeyuan Allen Zhu. 2010. Asymptotically optimal strategy-proof mechanisms for two-facility games. In *Proceedings of the 11th ACM conference on Electronic commerce*, pages 315–324.
- Hervé Moulin. 1980. On strategy-proofness and single peakedness. *Public Choice*, 35(4):437–455.
- Ariel D Procaccia and Moshe Tennenholtz. 2013. Approximate mechanism design without money. *ACM Transactions on Economics and Computation (TEAC)*, 1(4):1–26.
- Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, and 1 others. 2024. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652.
- Xin Sui, Craig Boutilier, and Tuomas Sandholm. 2013. Analysis and optimization of multi-dimensional percentile mechanisms. In *IJCAI*, pages 367–374. Cite-seer.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.
- Rui Zhang, Fei Liu, Xi Lin, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. 2024. Understanding the importance of evolutionary search in automated heuristic design with large language models. In *International Conference on Parallel Problem Solving from Nature*, pages 185–202. Springer.

A Extended Related Work

Mechanism Design for Facility Location. [Moulin \(1980\)](#) first developed a complete characterization of all strategyproof mechanisms for locating a facility where agents have single-peaked

preferences on a real line. [Procaccia and Tennenholtz \(2013\)](#) first considered strategyproof, approximately optimal mechanisms in facility location on the real line. For locating a single facility, they showed that locating the facility at the median agent location is the optimal mechanism for minimizing the total distance of all agents (i.e., the considered social cost with unweighted agents). For locating two facilities with n agents, ([Procaccia and Tennenholtz, 2013](#)) gave a lower-bound of $3/2 - O(1/n)$ and an upper-bound of $n - 2$ for deterministic strategyproof mechanisms. The lower bound was later improved to asymptotically and matching tight bounds of $\Omega(n)$ and $n - 2$ ([Lu et al., 2010](#); [Fotakis and Tzamos, 2014](#)). Beyond two facilities, all existing strategyproof mechanisms have an upper-bound of $\Omega(n)$ ([Fotakis and Tzamos, 2014](#); [Escoffier et al., 2011](#)). ([Golowich et al., 2018](#)) utilized neural networks to learn and design mechanisms for locating multiple facilities on the real line and optimizing the considered social cost with unweighted agents.

LLMs as Reasoning Agents. A plethora of work has shown that LLMs can act as autonomous agents capable of iterative reasoning and self-improvement. Chain-of-thought prompting ([Wei et al., 2022](#)) enables step-by-step reasoning, Re-Act ([Yao et al., 2023](#)) interleaves reasoning with action, and Reflexion ([Shinn et al., 2023](#)) enables agents to learn from past failures through verbal self-reflection. Our evolutionary framework can be viewed through a similar lens: The LLM reflects on parent mechanisms (via variation prompts) to generate improved candidates, and the prompt evolution component represents a meta-level of self-improvement where the LLM reasons about how to improve its own instructions.

B Code Template

```
def get_locations(samples):
    """
    Determines the optimal locations from a given list of
    location samples.

    Args:
    samples (list): A one-dimensional list containing the
    location samples.

    Returns:
    list: A one-dimensional list of the optimal locations,
    containing n_locations elements in [0,1].
    """

    # Placeholder (replace with your actual implementation)
    locations = ...

    return locations
```

Listing 1: Code template for unweighted settings.

```
def get_locations(samples, weights):
    """
    Determines the optimal locations from a given list of
    location samples.

    Args:
    samples (list): A one-dimensional list containing the
    location samples.
    weights (list): A list of fixed weights assigned to the
    samples: [5,1,1,1,1]

    Returns:
    list: A one-dimensional list of the optimal locations,
    containing n_locations elements in [0,1].
    """

    # Placeholder (replace with your actual implementation)
    locations = ...

    return locations
```

Listing 2: Code template for weighted settings, shown here for the case of $n = 5$, fixed weights.

C Prompts

C.1 Initialization Prompt

Figure S7 illustrates the exact initialization prompt.

Prompt for Initialization

I need help design a strategy to determine K facility locations in $[0,1]$ given a list of location samples. The objective is to minimize the sum of cost values to closest facility.

First, describe your new strategy and main steps in one sentence. The description must be inside a brace. Next, implement it in Python using the following template:

[Code Template]

Do not give additional explanations and do not use additional other packages.

Figure S7: Prompt used for initialization. See Appendix B for the designed “Code Template”.

C.2 Variation Prompts

Figure S8 illustrates the initial prompts for exploration and modification.

C.3 Prompt Evolution

Figure S9 illustrates the exact prompt used for evolving prompt strategies.

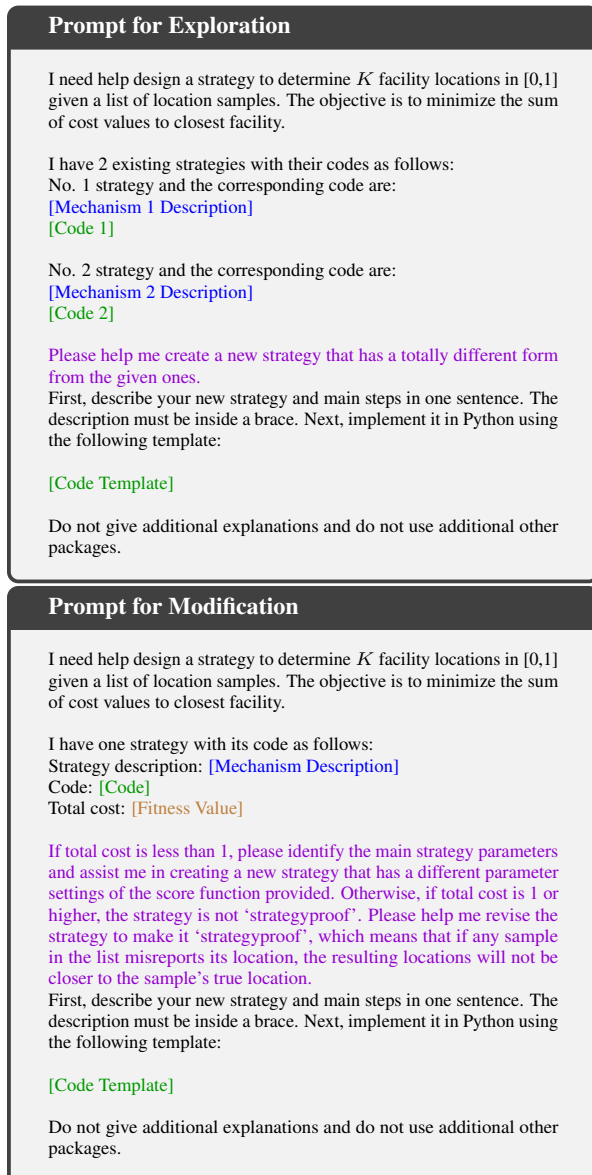


Figure S8: Prompts used for exploration and modification. The prompt strategies are marked in purple. See Appendix B for the designed “Code Template”.

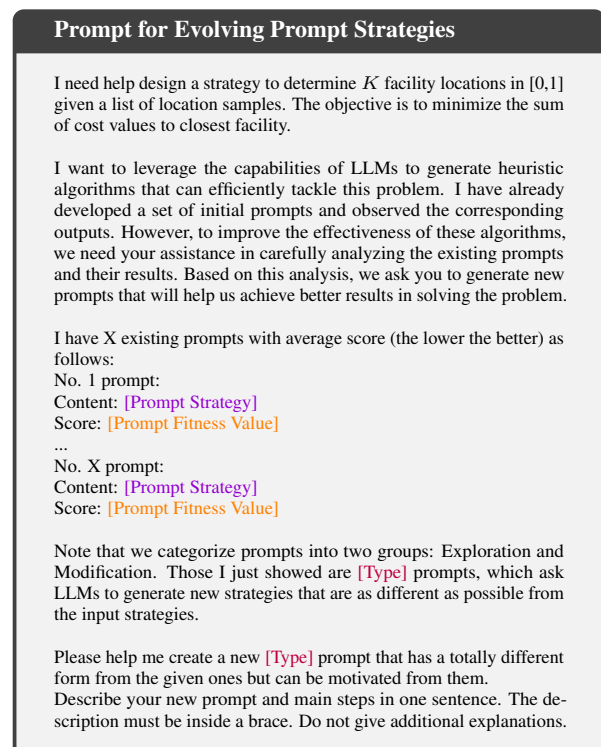


Figure S9: Prompts used for evolving prompt strategies, where [\[Type\]](#) is ‘Exploration’ or ‘Modification’ and $X = |\mathcal{P}| \leq N_p$.

D Designed Mechanisms

```
def get_locations(samples, weights):
    '''
    Determines the optimal locations from a given list of
    location samples.

    Args:
    samples (list): A one-dimensional list containing the
        location samples.
    weights (list): A list of fixed weights assigned to the
        samples: [5,5,1,1,1,1,1,1,1,1]

    Returns:
    list: A one-dimensional list of the optimal locations,
        containing n_locations elements in [0,1].
    '''
    n_locations = 4
    weights = [5, 5, 1, 1, 1, 1, 1, 1, 1, 1]

    # Combine samples and weights into a single list based on
    # weights
    weighted_samples = sorted(zip(samples, weights))
    sample_weighted = [sample for sample, weight in
        weighted_samples for _ in range(weight)]

    # Perform K-means clustering by selecting centroids of
    # grouped samples
    step = len(sample_weighted) // n_locations +
        (len(sample_weighted) % n_locations > 0)
    locations = [sample_weighted[i * step + step // 2] for i
        in range(n_locations)]

    return locations
```

Listing 3: Learnerd mechanism for $n = 10, K = 4$, weighted agents with normally distributed peaks.

E Complete Implementation Details

All experiments were conducted under Ubuntu 20.04 on a Linux virtual machine equipped with NVIDIA GeForce RTX 3050 Ti GPU and 12th Gen Intel(R) Core(TM) i7-12700H CPU @2.3GHz. We chose GPT-4o mini as our pretrained LLM. Our code in Python 3.10 is available at <https://github.com/AnonyMouse3005/LLMMech>.

We run three trials of LLMMech for each problem configuration, and three trials of LLMMech-e for each LLMMech trial (i.e., evolving directly from the initial population of LLMMech). In both versions, the temperature is fixed at 1. The evaluation budget T_e is defined in terms of the maximum number of generations, for which we set to 20 for all problems. The population size is 16, and the maximum running time for each instance is 60 seconds (except when $n = 25$, for which we increase to 180). Unless otherwise stated, we assign the threshold ε defined in Equation 2 as the max regret returned by RegretNet during training (the lowest one across different parameter initializations), which is typically small as demonstrated in Golowich et al. (2018).

For LLMMech-e, we set the size of top individuals in the heuristic population, b , to 3; the number

of max consecutive generations without improvement (for detecting stagnation), T_p , to 3; the size of the top individuals produced by a prompt strategy (for measuring its fitness), d , to 3; and the size of the prompt population to 5. During prompt evolution, we notice more complex prompt strategies may return highly-inefficient mechanisms. Therefore, for quality control of the LLM-generated prompts, we only admit one to the prompt strategy population if at least one of its produced mechanisms is valid, i.e., not experiencing timeout.

Note that beside ε , all above choices were made solely to ensure reasonable runtime (typically half an hour for LLMMech and a third quarter to an hour for LLMMech-e) without hampering the generation of nontrivial mechanisms. We believe the presented results for LLM methods can be further improved if we increase the scale of evolution.

F Resource Consumption

For LLMMech, the total number of LLM queries is [number of generations] $\times$ [number of prompt strategies] $\times$ [population size] = $20 \times 2 \times 16 = 640$. The same applies to LLMMech-e when there is no stagnation during the search process. Whenever stagnation occurs, an additional $2 + 2 \times$ [population size] = 34 LLM queries are incurred for generating two new prompt strategies (of exploration and modification types as shown in Figure S9) and initializing their respective offspring mechanisms. This prompt evolution process is called at most $\lfloor 20/3 \rfloor = 6$ times (i.e., the number of generations divided by the number of maximum consecutive generations without improvement in fitness values among the top- b mechanisms), which we found to be no more than 3 in all experiments. Since the number of prompt strategies now increases from originally 2 to at most 5 (i.e., the size of the prompt population), in the worst-case scenario where stagnation occurs as early as possible in the evolution for a maximum total of 3 times (hence prompt evolution is invoked at the end of the 4th, 7th, and 10th generations), the total number of LLM queries in chronological order is

$$\begin{aligned}
 & \underbrace{4 \times 2 \times 16}_{\text{first 4 gens.}} + \underbrace{2 + 2 \times 16}_{\text{first prompt evo.}} + \underbrace{3 \times 4 \times 16}_{\text{next 3 gens.}} \\
 & + \underbrace{2 + 2 \times 16}_{\text{second prompt evo.}} + \underbrace{3 \times 5 \times 16}_{\text{next 3 gens.}} + \underbrace{2 + 2 \times 16}_{\text{third prompt evo.}} \\
 & \quad + \underbrace{10 \times 5 \times 16}_{\text{last 10 gens.}} = 1462.
 \end{aligned}$$

In practice, prompt evolution occurs halfway through the evolution and hence a realistic estimate for LLMMech-e ranges from

$$\underbrace{10 \times 2 \times 16}_{\text{first 10 gens.}} + \underbrace{2 + 2 \times 16}_{\text{first prompt evo.}} + \underbrace{10 \times 4 \times 16}_{\text{last 10 gens.}} = 994$$

to

$$\begin{aligned} & \underbrace{10 \times 2 \times 16}_{\text{first 10 gens.}} + \underbrace{2 + 2 \times 16}_{\text{first prompt evo.}} + \underbrace{3 \times 4 \times 16}_{\text{next 3 gens.}} \\ & + \underbrace{2 + 2 \times 16}_{\text{second prompt evo.}} + \underbrace{3 \times 5 \times 16}_{\text{next 3 gens.}} + \underbrace{2 + 2 \times 16}_{\text{third prompt evo.}} \\ & \quad + \underbrace{4 \times 5 \times 16}_{\text{last 4 gens.}} = 1174 \end{aligned}$$

for 1 and 3 calls of the prompt evolution process, respectively.

The longest input prompts for exploration and modification from our experiments require 800 and 600 tokens (using the converter from <https://platform.openai.com/tokenizer>), respectively, and the longest output returned from the employed LLM (GPT-4o mini) containing the mechanism description and its corresponding code has 300 tokens. Based on our previously derived formula for the number of LLM queries for LLMMech, at the current rate of $c_i = \$0.150/1\text{M}$ input tokens and $c_o = \$0.600/1\text{M}$ output tokens for API calling of GPT-4o mini (<https://openai.com/api/pricing/>), the estimated cost of LLMMech is hence $20 \cdot 16 \cdot [(800 \cdot c_i + 300 \cdot c_o) + (600 \cdot c_i + 300 \cdot c_o)] = \$0.18$.

We now elaborate on the number of LLM queries for LLMMech-e. Whenever stagnation occurs, an additional $2+2 \times [\text{population size}] = 34$ LLM queries are incurred for generating two new prompt strategies and initializing their respective offspring mechanisms, and subsequent generations have more prompt strategies (up to 5 i.e., the chosen size of the prompt population) for generating offspring mechanisms. This prompt evolution process is called at most 3 times from our observations and typically first occurs halfway through the evolution (i.e., around the 10th generation). Hence, the max number of LLM queries for LLMMech-e is 1174. Given that the longest prompt for evolving prompt strategies requires 650 tokens, and the longest new prompt strategy returned from LLM has 50 tokens, LLMMech-e thus costs at most \$0.33.

G Additional Experiments

G.1 Main Results

Weighted Social Cost with Arbitrary Weights.

We report the sets of arbitrary weights used in Table S1. They were sampled uniformly from $\{1, \dots, 5\}$. Figure S10 shows results for $n = 5$ and 10.

#	$n = 5$	$n = 10$
1	5 3 3 3 1	5 4 4 3 2 2 2 1 1 1
2	4 4 4 3 3	5 5 4 3 2 2 2 1 1 1
3	3 3 3 2 1	5 5 4 3 3 2 2 2 1 1
4	5 2 1 1 1	5 5 4 4 4 3 2 2 1 1
5	5 4 3 3 2	4 4 4 4 4 4 4 3 2 2
6	4 4 4 3 1	5 4 4 3 3 2 1 1 1 1
7	4 2 2 2 1	5 5 4 3 3 3 3 2 2 1
8	5 4 2 2 1	5 4 4 3 3 3 3 2 2 1
9	5 4 4 3 2	5 5 5 5 4 4 4 4 4 4
10	5 4 4 3 3	5 5 5 4 4 3 2 2 2 1

Table S1: Set of arbitrary weights used.

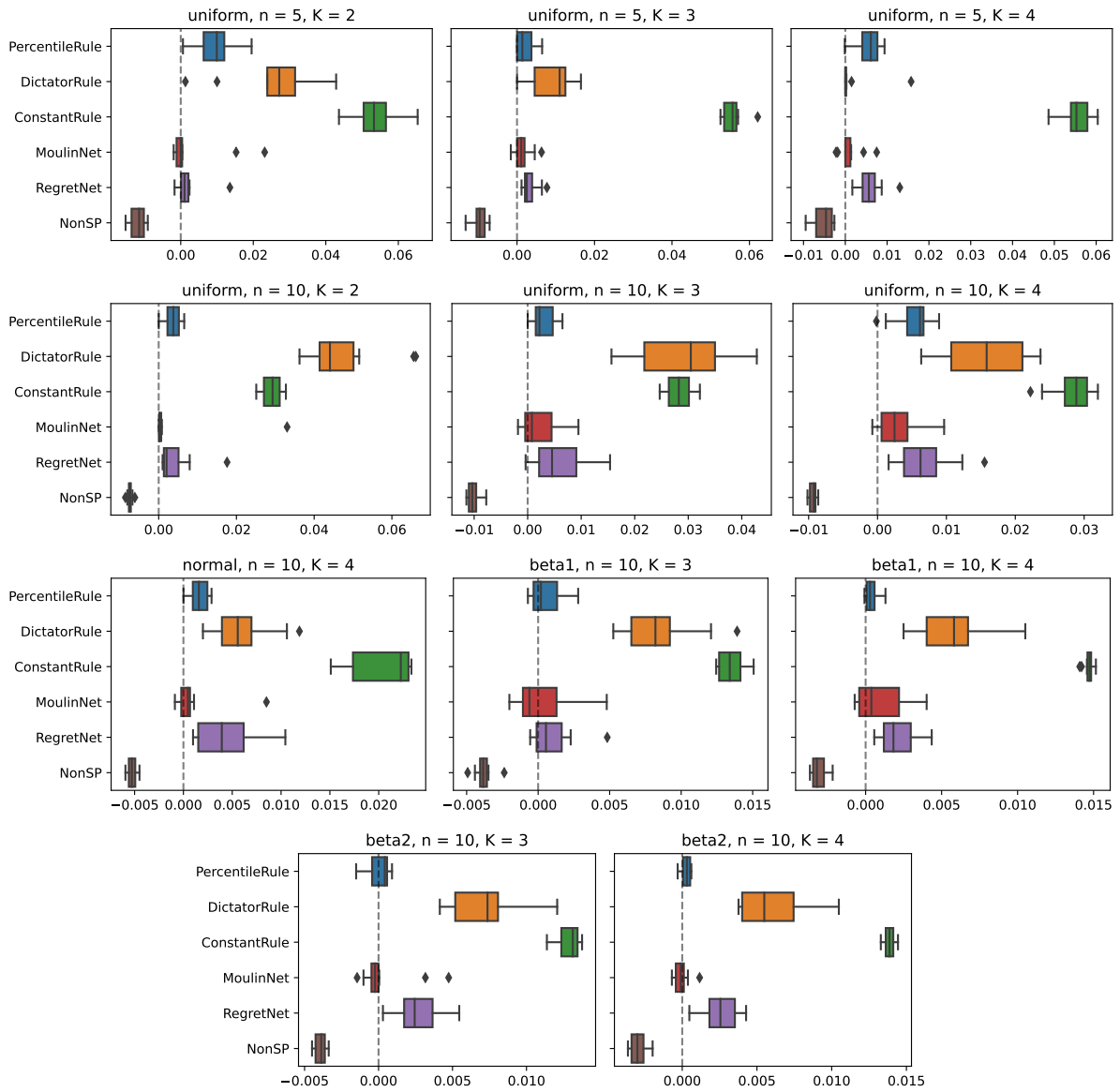


Figure S10: Results for net difference with respect to LLMMech-e in (arbitrarily) weighted social cost.

Unweighted Social Cost. Figure S11 shows results for $n = 5, 10, 25$, each with $K \in \{1, 2, 3, 4\}$ (we additionally consider $K = 5$ for $n = 25$).



Figure S11: Results for $n = 5$ (first two rows), $n = 10$ (next two rows), and $n = 25$ (bottom two rows), unweighted social cost. Values on top of bars (if any) represent regret (max across agents) incurred by the mechanism. The dashed lines display values from NonSP. (u—uniform, n—normal, b1—beta1, b2—beta2.)

G.2 Generalizability Results

Out-of-Distribution Performance (Weighted).

Figure S12 shows results for weighted $n = 5$ and 10, each with $K \in \{1, 2, 3, 4\}$, when baselines are also evaluated on OOD peaks.

Generalizability to Larger Instances. We examine the generalizability of LLM-generated mechanisms to larger instances with more agents. Note that this experiment is reasonable for unweighted settings only. Figure S13 shows results for $n = 10$ and $n = 25$, where LLM methods are trained on $n = 5$ instances. We observe impressive generalization performance from both LLMMech and LLMMech-e, with social costs generally close to or in some settings (e.g., ‘u_n10_K4’ and ‘b1_n10_K4’) even better than other methods³. This overall generalizability allows the learned mechanisms to quickly adapt to larger problems with minimal computational overheads. In contrast, the neural networks need to be retrained from scratch and are highly inefficient for large input size (i.e., n), whereas the manually designed rules also require recomputation, which are as costly given their brute-force nature.

³Handcrafted rules and neural networks are restricted to fixed input size and hence only applicable to similar-sized instances.

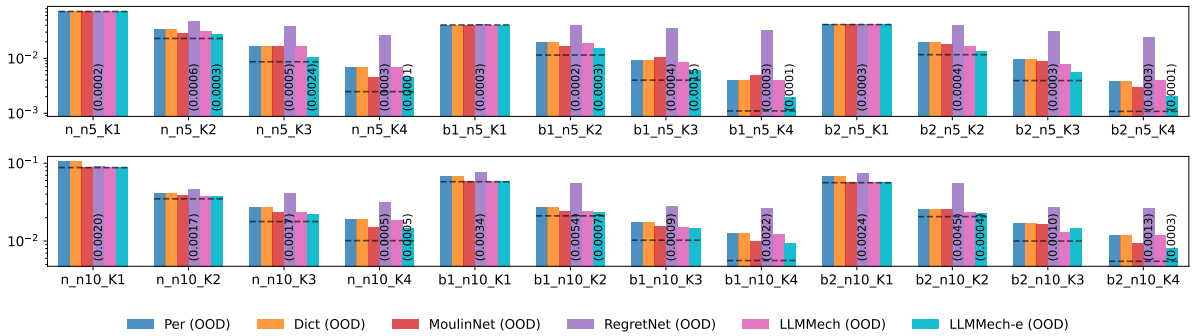


Figure S12: Out-of-distribution (OOD) results for $n = 5$ (top row) and $n = 10$ (bottom row), weighted social cost. Values on top of bars (if any) represent regret (max across agents) incurred by the mechanism. The dashed lines display values from NonSP. (n—normal, b1—beta1, b2—beta2.)

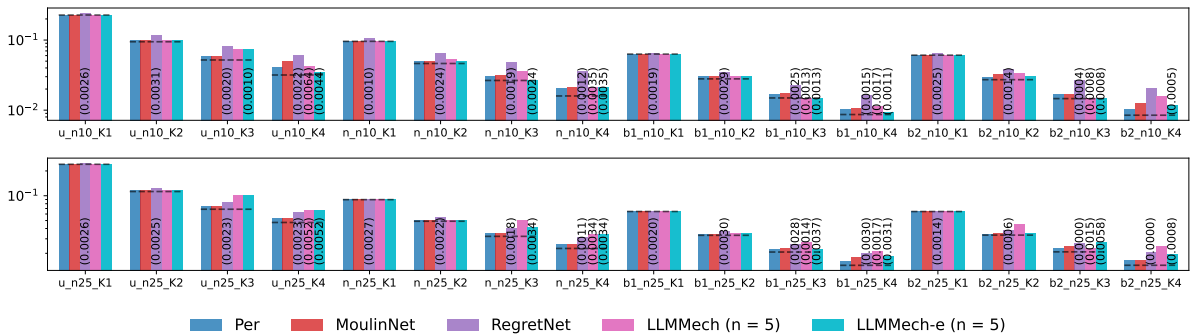


Figure S13: Results for generalizability of LLM-generated mechanisms from $n = 5$ to (top) $n = 10$ and (bottom) $n = 25$, unweighted social cost. Baselines were trained directly on $n = 10, 25$ datasets, respectively. Values on top of bars (if any) represent regret (max across agents) incurred by the mechanism. The dashed lines display values from NonSP. (u—uniform, n—normal, b1—beta1, b2—beta2.)

G.3 Comparison with Classical Automated Mechanism Design

Classical AMD discretizes the type space and solves the resulting optimization problem exactly (Conitzer and Sandholm, 2002, 2004) via linear programming (LP). For completeness, we compare this approach against our reported results.

Setup. We use the smallest multi-facility configuration appearing in Figure 2, namely ‘u_n5_K2’ ($n = 5$, $K = 2$, uniform peaks, fixed weights), together with the same 1,000 evaluation profiles and the same misreport budget $B = 10$ as everywhere else. We discretize $\Omega = [0, 1]$ into m equally spaced grid points. The LP minimizes the expected weighted social cost over randomized mechanisms: One lottery variable per pair of type profile and facility placement, i.e., $m^n \binom{m+K-1}{K}$ variables, subject to $n(m-1)m^n$ strategyproofness constraints. Randomized allocations are unavoidable here because computing an optimal deterministic mechanism is NP-hard (Conitzer and Sandholm, 2004). The solution is exactly strategyproof, but it only accepts reports lying exactly on the grid. Therefore, when evaluating it on real-valued profiles, we round each reported peak to the nearest grid point to index the lottery, while computing social cost and regret at the original, unrounded peaks.

Results. The resulting mechanism is subpar on all three axes we care about.

(1) *Social Cost.* With $m = 5, 6, 7$ grid points, the rounded LP attains weighted social cost of 0.091, 0.080, and 0.073, respectively. Even its best value trails every learned mechanism in this block of Figure 2 (LLMMech-e 0.052, MoulinNet 0.054, LLMMech 0.061, RegretNet 0.063) and the best dictatorial rule (0.070), and it beats the best percentile rule (0.081) only at $m \geq 6$.

(2) *Strategyproofness.* The empirical regret from the rounded LP mechanism is about 0.01 on average, which is on the order of the social cost itself. For reference, RegretNet incurs 0.0003 on the same setting, and both LLMMech and LLMMech-e attain zero empirical regret. Understandably, as noted in the AMD survey of Curry et al. (2026), mapping reports back to the original continuous space “introduces violations of the incentive compatibility (i.e., strategyproofness) constraint”.

(3) *Scalability.* Already at $m = 8$, the program has about 1.2×10^6 variables and 1.1×10^6 constraints, which exhausts the memory of our machine (speci-

fications in Appendix E) during Gurobi’s presolve. Since the growth is exponential in n , refining the grid enough to close the social cost gap is not an option.

G.4 Robustness to a Larger Misreport Budget

The empirical regret is computed with a misreport budget of $B = 10$ draws per agent per profile, following the evaluation protocol of Golowich et al. (2018), which exposes each generated mechanism to a total of $1000 \times 10 \times n$ misreports in total. To strengthen our claims regarding empirical strategyproofness, we re-evaluate every mechanism reported in this paper under a larger budget of $B = 100$ misreports per agent per profile, holding the evaluation profiles and all other settings fixed. The increased budget leaves the picture intact, as we observe the max regret consistently stays at zero wherever it was zero, and rises negligibly elsewhere, i.e., two orders of magnitude below the corresponding social cost.

G.5 Ablation Studies and Further Analysis

Does the LLM’s Understanding Matter? To isolate the contribution of the LLM’s code understanding and generation capabilities from the evolutionary search framework itself, we compare LLM-Mech against a **random mutation baseline** (RandMut) that replaces the LLM with a simple program perturbation operator. Specifically, RandMut generates offspring mechanisms by randomly modifying numerical constants (sampled uniformly from $[0, 1]$), swapping sorting orders, or inserting/removing list operations in the parent mechanism’s code. The evolutionary framework—including fitness evaluation, selection, and population management—remains identical.

Table S2 reports results on representative settings with $n = 10$ agents and fixed weights. RandMut produces valid (non-erroring) mechanisms in only 38% of attempts, compared to 91% for LLMMech. Among valid mechanisms, RandMut achieves substantially higher social costs: When averaged across all $K \in \{1, 2, 3, 4\}$ on the uniform distribution, RandMut’s social cost is 0.0842, compared to 0.0341 for LLMMech and 0.0298 for LLMMech-e. Moreover, only 24% of RandMut’s surviving mechanisms achieve zero empirical regret, versus 87% for LLMMech. This confirms that the LLM’s ability to reason about the structure of mechanisms—rather than merely serving as a stochastic perturbation source—is essential to the

framework’s success.

Setting	RandMut	LLMMech	LLMMech-e	NonSP
u_n10_K2	0.0614 _(0.0031)	0.0283 _(0.0000)	0.0261 _(0.0000)	0.0247
u_n10_K3	0.0793 _(0.0048)	0.0312 _(0.0000)	0.0275 _(0.0000)	0.0198
u_n10_K4	0.1119 _(0.0072)	0.0427 _(0.0010)	0.0357 _(0.0007)	0.0153
n_n10_K3	0.0507 _(0.0039)	0.0218 _(0.0000)	0.0196 _(0.0006)	0.0112

Table S2: Weighted social cost (and max regret in parentheses) for representative settings. RandMut replaces the LLM with random code perturbation. Lower is better.

Sensitivity to the Choice of LLM. We investigate whether LLMMech’s performance depends critically on the specific LLM used. We replace GPT-4o mini with **Llama-3-8B-Instruct**, an *open-weight* model of comparable scale, and re-run LLMMech (without prompt evolution) on the uniform distribution settings with $n = 10$.

Table S3 shows that Llama-3-8B-Instruct produces valid mechanisms at a lower rate (76% vs. 91% for GPT-4o mini), primarily due to more frequent formatting errors in the generated code (e.g., missing return statements, incorrect function signatures). Among valid mechanisms, the social costs are 8–15% higher than those achieved with GPT-4o mini, but still substantially outperform the best handcrafted rules (PercentileRule) in most settings and remain competitive with MoulinNet. Notably, the mechanisms generated by Llama-3-8B-Instruct are qualitatively similar in structure to those from GPT-4o mini—for $K = 1$, both models consistently rediscover variants of the median rule—suggesting that the evolutionary framework is partially robust to LLM quality.

Setting	GPT-4o mini	Llama-3-8B	Per.	MoulinNet
u_n10_K1	0.0198	0.0203	0.0198	0.0201
u_n10_K2	0.0283	0.0319	0.0364	0.0296
u_n10_K3	0.0312	0.0351	0.0392	0.0334
u_n10_K4	0.0427	0.0486	0.0518	0.0453

Table S3: Weighted social cost of LLMMech with different backbone LLMs on uniform distribution, $n = 10$. All mechanisms shown have max regret less than ε .

Error Analysis of LLM Generations. Across all experiments, the LLM generates candidate mechanisms that fall into four categories: (1) *Valid and strategyproof* (71% of all generations with GPT-4o mini): The mechanism runs without error and achieves empirical max regret below ε . These enter the population. (2) *Valid but not strategyproof* (16%): The mechanism executes correctly

but yields max regret above ε , receiving a fitness penalty. Common patterns include mechanisms that naïvely optimize social cost without considering strategic incentives—e.g., placing facilities at the sample mean of agent locations, which is manipulable. (3) *Runtime errors* (9%): The generated code fails during execution. The most frequent errors are index-out-of-bounds errors when the mechanism assumes a fixed number of agents (4.2%), division-by-zero in normalization steps (2.8%), and incorrect use of imported libraries (2.0%). (4) *Time-out* (4%): The mechanism runs but exceeds the 60-second time limit, typically due to iterative optimization loops that do not converge within the budget.

We observe that the fraction of valid and strategyproof mechanisms increases over the course of evolution: from approximately 42% in the first generation to 83% by generation 20. This suggests that the selection pressure in the evolutionary framework, combined with the LLM’s ability to learn from parent mechanisms provided in the prompt, effectively guides the search toward the feasible region.

Impact of Prompt Evolution. To isolate the contribution of prompt evolution (Section 4.4), we compare LLMMech (fixed prompts) against LLMMech-e (with prompt evolution) across all 32 weighted settings ($n \in \{5, 10\}$, $K \in \{1, 2, 3, 4\}$, four distributions). LLMMech-e achieves equal or lower social cost than LLMMech in 28 out of 32 settings, with the improvement concentrated in higher- K settings ($K \geq 3$) where the search space is larger and premature convergence is more likely. The average improvement of LLMMech-e over LLMMech across $K = 4$ settings is 11.3%, compared to only 2.1% for $K = 1$ settings where the median rule is already near-optimal and easily discovered by either variant.

We also find that prompt evolution is triggered between 1 and 3 times per run (across all experiments), typically occurring between generations 7 and 13. The newly generated prompt strategies often introduce qualitatively different search directions—for example, the prompt in Figure 6 shifts the focus from parameter tuning to a holistic analysis of the cost-strategyproofness trade-off, which led to the discovery of the K-means-based mechanism described in Figure 4.