CVaR-RMD: CVaR Selection Pressure for Robust Metaheuristic Discovery

Ruodai Cui[1,2] **Li Niu**[3]*

[1]Qualcomm Technologies, Inc. [2]The Hong Kong Polytechnic University
[3]Department of Computer Science and Engineering, MoE Key Lab of Artificial Intelligence, Shanghai Jiao Tong University
ruodcui@connect.polyu.hk ustcnewly@sjtu.edu.cn

Abstract

Large language models have recently been used to discover heuristics through iterative program search, offering a promising approach to automated algorithm design. However, candidates selected by average performance may degrade across instance families, random seeds, and problem scales because mean-based objectives can obscure rare but severe failures. We propose CVaR-RMD, a risk-sensitive evolutionary framework that uses an LLM to mutate executable heuristic programs and ranks candidates by empirical conditional value-at-risk over heterogeneous evaluation conditions. Across three combinatorial optimization domains, CVaR-based selection improves upper-tail performance and reduces failure rates while maintaining competitive mean performance. These gains persist on larger unseen instances and independently curated within-domain benchmarks. Code is available *here*.

1 Introduction

Heuristics and metaheuristics are central to combinatorial, stochastic, black-box, or partially specified optimization problems. Their practical value depends not only on benchmark performance during algorithm discovery but also on whether it persists when evaluation conditions change within the same domain. Such changes include shifts across instance families, problem scales, and random seeds. We call this setting *within-domain transfer* under benchmark distribution shift: the optimization problem and solution representation remain unchanged, but benchmark conditions differ between discovery and testing. This concern is longstanding in heuristic optimization, where empirical success often depends on interactions among algorithm design, benchmark distributions, and stochastic variation (Talbi, 2009; Hoos and Stützle, 2004; Blum and Roli, 2003).

Automated algorithm design naturally supports studying this challenge because candidates are generated and selected by their observed performance under specified evaluation conditions. However, searching algorithmic choices alone does not ensure within-domain transfer, which depends critically on the evaluation conditions and ranking objective. Hyper-heuristics automate design through search spaces of heuristics or heuristic components, while algorithm configuration and selection evaluate candidate procedures across instance sets to support performance beyond individual discovery instances (Burke et al., 2013; Rice, 1976; Hutter et al., 2011; López-Ibáñez et al., 2016; Ansótegui et al., 2009).

A candidate heuristic may achieve strong average performance yet fail on rare instances, different stochastic seeds, or larger scales. This issue is especially relevant to LLM-generated metaheuristics because generated code may exploit a fixed evaluation protocol, such as a narrow instance generator, few seeds, or a restricted scale range. The outer loop may therefore select heuristics with high expected performance but unstable tails. In evolutionary search, this selection pressure rewards favorable means even when outcome distributions contain severe failures (Bäck, 1996; Eiben and Smith, 2003; Goldberg and Deb, 1991).

Because such failures occur only under adverse subsets of evaluation conditions, mean aggregation may obscure them. A useful selection criterion should therefore emphasize adverse outcomes without allowing a single extreme observation to dominate candidate ranking. CVaR satisfies this requirement by averaging performance over a configurable worst-performing fraction of evaluation outcomes. We therefore study a minimal change to discovery: replacing mean-based fitness with CVaR-based fitness while leaving the outer loop unchanged. Each candidate heuristic is evaluated over conditions spanning instance families, stochas-

*Corresponding author.

Findings of the Association for Computational Linguistics: EMNLP 2026, pages 14407–14429
October 24-29, 2026 ©2026 Association for Computational Linguistics

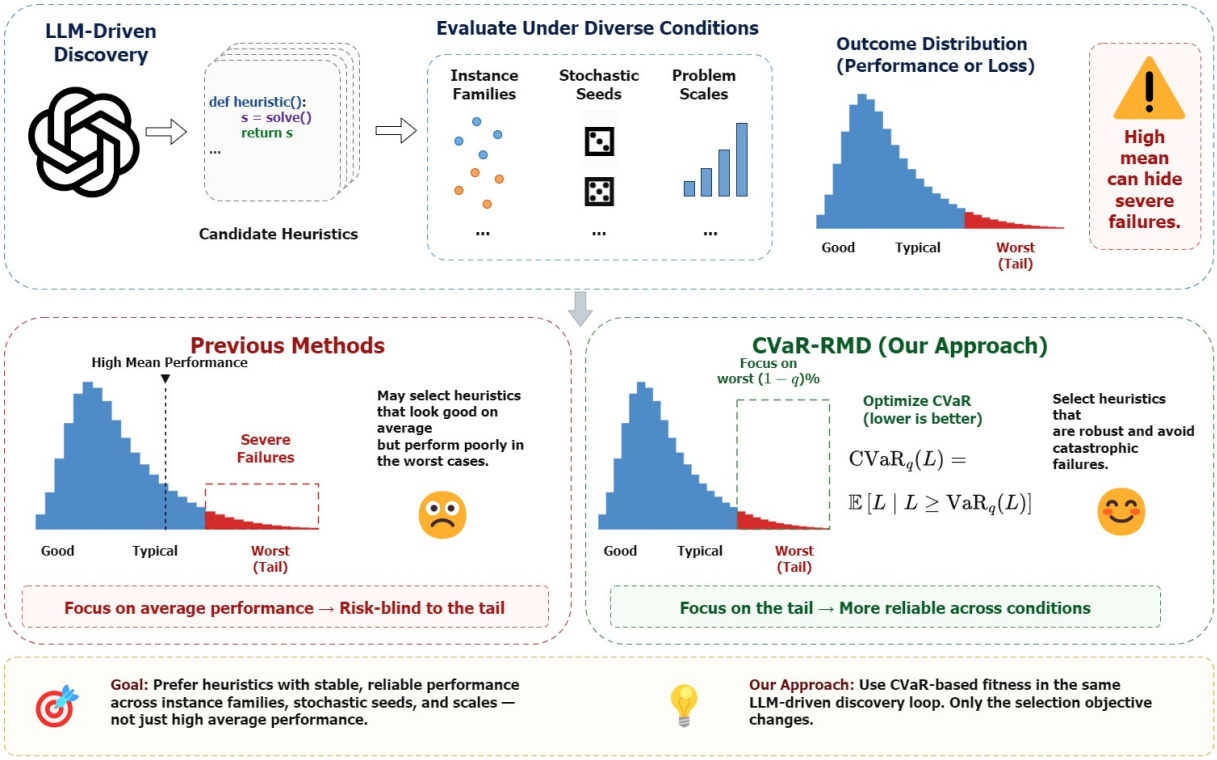


Figure 1: Motivation for CVaR-RMD. Mean-based fitness in LLM-driven metaheuristic discovery rewards strong average performance while tolerating severe failures on rare instances or seeds. CVaR shifts selection pressure toward worst-quantile behavior, discouraging heuristics that depend on favorable evaluation conditions.

tic seeds, and scales. Its fitness uses conditional value-at-risk (CVaR), a coherent risk measure estimating expected loss within the worst fraction of outcomes at a chosen quantile level (Artzner et al., 1999; Rockafellar and Uryasev, 2000, 2002; Pflug, 2000). The resulting CVaR-based Robust Metaheuristic Discovery framework (CVaR-RMD) uses the LLM as a mutation generator to propose heuristic code, while estimated CVaR fitness determines candidate acceptance. As Figure 1 shows, mean-based selection tolerates severe tail failures that CVaR-RMD penalizes. Because selected programs become parents and prompt context, CVaR-based ranking affects both final selection and the subsequent trajectory of LLM-driven program discovery.

This formulation shifts selection toward worst-quantile behavior and discourages candidates dependent on favorable evaluation conditions. It follows robust and distributionally robust optimization, where objectives accounting for adverse outcomes and distributional variation replace or supplement expectation-based criteria (Ben-Tal et al., 2009; Beyer and Sendhoff, 2007; Bertsimas et al., 2011; Rahimian and Mehrotra, 2019; Duchi and Namkoong, 2021). It also aligns with optimization under noise and uncertainty, which uses repeated evaluation and robust criteria to reduce the selection of brittle solutions (Jin and Branke, 2005; Branke et al., 2008; Beyer and Sendhoff, 2007). Our empirical analysis tests whether risk-based selection improves heuristic stability and transfer across evaluation conditions without adding architectural components to the LLM-driven discovery pipeline.

The contributions of this work are as follows:

1. We introduce CVaR-based selection into LLM-guided metaheuristic discovery, using empirical CVaR as the primary fitness criterion to emphasize adverse-quantile behavior across heterogeneous evaluation conditions.

2. We propose CVaR-RMD, an execution-driven, risk-sensitive framework that integrates CVaR-based candidate ranking, accumulated execution outcomes, and tail-case feedback into an evolutionary program-discovery loop for generating robust executable heuristics.

3. We conduct controlled, budget-matched experiments across three combinatorial optimization domains and show that CVaR-RMD improves upper-tail performance and reduces ex-

ecution failures while maintaining competitive mean performance.

2 Related Work

Automated Heuristic Design Automated heuristic design has been studied through hyper-heuristics, algorithm selection/configuration, and program synthesis. Hyper-heuristics search over heuristic spaces by selecting existing low-level operators or generating new components that generalize across instances (Burke et al., 2013; Cowling et al., 2001; Ochoa et al., 2012). Algorithm selection and configuration address related questions from a statistical and empirical perspective, using instance features, performance observations, and iterative evaluation to identify robust algorithms or parameterizations under limited budgets (Rice, 1976; Hutter et al., 2009, 2011; López-Ibáñez et al., 2016; Ansótegui et al., 2009). Earlier genetic programming and grammatical evolution methods directly searched over program structures via fitness-based selection and variation (Koza, 1992; O'Neill and Ryan, 2001; Langdon and Poli, 2002). These works frame heuristic construction as a search problem whose evaluation protocol and selection objective strongly affect generalization.

Recent LLM-based methods extend this paradigm by coupling generative proposal mechanisms with external evaluation. FunSearch evolves and validates LLM-generated code variants using task-specific scoring functions (Romera-Paredes et al., 2024). EoH co-evolves natural-language "thoughts" and executable heuristics (Liu et al., 2024a), while ReEvo uses LLMs as hyper-heuristics with reflective evolution (Ye et al., 2024). LLaMEA evolves LLM-generated metaheuristics for black-box continuous optimization (van Stein and Bäck, 2024), and LLAMBO integrates LLM reasoning into Bayesian optimization pipelines (Liu et al., 2024b). Despite differences in representation, prompting, and search dynamics, these systems typically depend on scalar fitness values from limited empirical evaluations.

Robust and Risk-Aware Optimization Robust optimization seeks solutions that remain effective under uncertainty, emphasizing robustness metrics and the trade-off between nominal performance and stability (Ben-Tal et al., 2009; Beyer and Sendhoff, 2007; Bertsimas et al., 2011). Distributionally robust optimization further optimizes against worst-case distributions within ambiguity sets (Delage and Ye, 2010; Rahimian and Mehrotra, 2019; Duchi and Namkoong, 2021). Among risk measures, conditional value-at-risk (CVaR) captures tail behavior and supports tractable sample-based formulations (Artzner et al., 1999; Rockafellar and Uryasev, 2000, 2002; Pflug, 2000). Such measures are widely used in finance, stochastic programming, reinforcement learning, and decision-making (Shapiro et al., 2009; Tamar et al., 2015; Chow et al., 2015). Robustness under noise and uncertainty has also been extensively studied in evolutionary computation (Jin and Branke, 2005; Branke et al., 2008; Beyer and Sendhoff, 2007). These studies motivate objectives that account for unfavorable outcomes rather than only average performance.

Benchmarking, Robustness, and Transfer Evaluation protocols are central to empirical combinatorial optimization because conclusions about algorithm performance depend strongly on the instances and conditions under which algorithms are assessed. Benchmark suites and heterogeneous instance collections are commonly used to evaluate whether algorithms remain effective beyond their development conditions (Johnson and McGeoch, 1997; Ochoa et al., 2012; Reinelt, 1991). In routing and learned combinatorial optimization, prior studies have documented performance degradation when methods are evaluated across problem scales, instance generators, and instance families that differ from those used during development (Vinyals et al., 2015; Bello et al., 2017; Kool et al., 2019; Joshi et al., 2019). These findings motivate evaluation protocols that account not only for average performance but also for robustness under heterogeneous and shifted benchmark conditions. When candidate performance is summarized solely by the empirical mean, the selection process may favor heuristics that perform well on typical instances while remaining vulnerable to rare, difficult, or shifted cases. This concern is particularly relevant to LLM-driven heuristic discovery, where finite empirical evaluations repeatedly determine which generated programs are retained and used to guide subsequent search (Romera-Paredes et al., 2024; Liu et al., 2024a; Ye et al., 2024; van Stein and Bäck, 2024). Robust evaluation and selection criteria are therefore important for discovering heuristics that maintain reliable performance across instance families, random seeds, and problem scales.

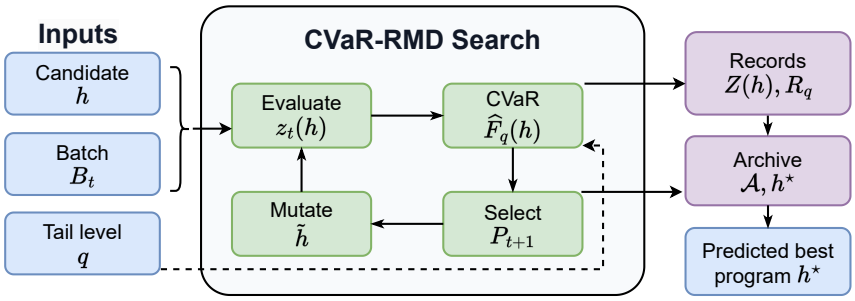


Figure 2: Architecture of the CVaR-guided program search method. Notations are defined in Sec. 3.

3 Method

We seek an executable heuristic program for a family of optimization instances. Let $\mathcal{I}$ be the instance space, $\mathcal{D}$ the scale-parameter set, and $\mathcal{S}$ the random-seed set controlling stochastic execution. An evaluation environment is $e = (i, d, s) \in \mathcal{E}$, where $i \in \mathcal{I}$, $d \in \mathcal{D}$, $s \in \mathcal{S}$, and $\mathcal{E} = \mathcal{I} \times \mathcal{D} \times \mathcal{S}$. Environments are sampled from a fixed distribution P over $\mathcal{E}$. A candidate heuristic is a program $h \in \mathcal{H}$ implementing a prescribed callable interface. Given e, executing h returns a feasible solution $x = h(i, d, s)$ or a failure state. The scalar loss is $\ell(h; e) \in \mathbb{R}$, with lower values preferred. On success, $\ell(h; e)$ is computed from the task objective and returned solution. If the program fails to parse, compile, satisfy the interface, terminate within the execution limit, or return a feasible solution, the loss is set to a fixed penalty ℓ_{fail}.

Our objective accounts for the upper tail of the loss distribution rather than its mean alone. For $E \sim P$, let $L_h = \ell(h; E)$ be the random loss induced by h. Given tail level $q \in (0, 1)$, the population objective is

$$h^\star \in \arg\min_{h \in \mathcal{H}} \mathrm{CVaR}_q(L_h). \tag{1}$$

Here $\mathrm{CVaR}_q(L) = \inf_{\eta \in \mathbb{R}} \{\eta + \frac{1}{1-q}\mathbb{E}[(L-\eta)_+]\}$, where $(x)_+ = \max(x, 0)$. The instance distribution, callable interface, loss transformation, failure penalty, and evaluation budgets are fixed before search.

The method performs population-based program search with tail-risk scoring. At generation t, a population $\mathcal{P}_t = \{h_t^{(1)}, \ldots, h_t^{(N)}\}$ is evaluated on sampled environments. Each candidate yields a loss vector converted to empirical CVaR fitness. Candidates are ranked, selected programs are mutated by a language-model operator, and returned source code is parsed, compiled, validated, and evaluated. The next population is formed by elitist replacement over the union of parents and offspring, as illustrated in Figure 2. The remainder of this section is organized as follows: Section 3.1 defines the evaluation distribution and loss construction, Section 3.2 specifies empirical CVaR scoring, and Section 3.3 describes the evolutionary search loop.

3.1 Evaluation Distribution and Loss Construction

The evaluation module maps sampled problem settings to scalar losses. Its inputs are a candidate program h, batch size m_t, and environment distribution P; its output is

$$z_t(h) = \big(z_{t,1}(h), \ldots, z_{t,m_t}(h)\big) \in \mathbb{R}^{m_t},$$

with each entry corresponding to one execution of h on a sampled environment.

Instances are produced by a finite generator set $\mathcal{G}$. A generator $g \in \mathcal{G}$ maps a scale value d and instance seed u to $i = g(d, u)$. The environment distribution samples $g \sim \pi_{\mathcal{G}}$, $d \sim \pi_{\mathcal{D}}(\cdot \mid g)$, $u \sim \pi_{\mathcal{U}}$, and $s \sim \pi_{\mathcal{S}}$, then sets $i = g(d, u)$ and $e = (i, d, s)$. The generator identity, scale value, instance seed, and execution seed are stored with the environment, making each execution reproducible within the protocol.

At generation t, the sampled batch is $B_t = \{e_{t,1}, \ldots, e_{t,m_t}\}$. All candidates in the same generation use the same batch, so scores are comparable under identical environments. For $e_{t,j} = (i_{t,j}, d_{t,j}, s_{t,j})$, the harness calls the fixed interface and obtains a solution object $x_{t,j}$ or a failure state. A deterministic validator checks whether $x_{t,j}$ satisfies the feasibility constraints of $i_{t,j}$. If the program is valid and the solution feasible, the raw task value is transformed into a scalar loss. For minimization objective $a(i, x)$, $z_{t,j}(h) = a(i_{t,j}, x_{t,j})$. For

maximization reward $r(i, x)$, the equivalent minimization loss is $z_{t,j}(h) = -r(i_{t,j}, x_{t,j})$.

If a reference value $b(i)$ is available, a normalized minimization loss can be used:

$$z_{t,j}(h) = \frac{a(i_{t,j}, x_{t,j}) - b(i_{t,j})}{\max\{|b(i_{t,j})|, \epsilon\}}, \qquad \epsilon > 0.$$

Failure handling precedes score computation. If the source code cannot be parsed or compiled, all entries for that candidate in the current batch are set to ℓ_{fail}. If compilation succeeds but one execution fails, only that entry is set to ℓ_{fail}. Thus, the loss vector may contain objective-derived and penalty values. The scoring module receives only this scalar vector, not program text, exception type, or feasibility metadata.

The batch policy controls loss accumulation across generations. With a fixed batch, the same B is used throughout search, and each candidate is scored on identical environments whenever evaluated. With resampling, a new B_t is drawn each generation. In the accumulated-loss variant, the method stores a multiset $Z(h)$ for every evaluated program. When h is evaluated on a new batch,

$$Z(h) \leftarrow Z(h) \cup \{z_{t,1}(h), \ldots, z_{t,m_t}(h)\}.$$

The empirical score is recomputed from all entries in $Z(h)$. A candidate must contain at least $m_{\min}$ stored losses before entering the elite archive; until then, its score is provisional and used only for within-generation ranking.

3.2 Empirical CVaR Fitness

The scoring module maps a loss vector $z(h) = (z_1, \ldots, z_m)$ and tail level q to empirical CVaR fitness $\widehat{F}_q(h)$, along with the empirical tail threshold and identifiers of tail environments.

Let $\alpha = 1 - q$. The empirical score is the sample analogue of Eq. (1):

$$\widehat{F}_q(h) = \min_{\eta \in \mathbb{R}} \left\{ \eta + \frac{1}{\alpha m} \sum_{j=1}^{m} (z_j - \eta)_+ \right\}. \quad (2)$$

Since Eq. (2) is piecewise linear in η, an optimizer can be selected from observed losses. Let $z_{(1)} \leq \cdots \leq z_{(m)}$ be sorted losses, $r = \alpha m$, $k = \lceil r \rceil$, and $\rho = k - r$. The empirical CVaR is

$$\widehat{F}_q(h) = \frac{1}{r} \left(\sum_{a=m-k+2}^{m} z_{(a)} + (1 - \rho) z_{(m-k+1)} \right), \quad (3)$$

where an empty sum is zero. If r is an integer, $\rho = 0$, and Eq. (3) averages the largest r losses. If r is fractional, losses strictly above the boundary receive unit weight, and the boundary loss receives weight $1 - \rho$.

The empirical value-at-risk threshold is $\widehat{\eta}_q(h) = z_{(m-k+1)}$. The tail record stores indices of losses larger than $\widehat{\eta}_q(h)$ and enough boundary indices with value $\widehat{\eta}_q(h)$ to match the fractional boundary weight in Eq. (3). For each tail index, the method records generator identity, scale value, instance seed, execution seed, realized loss, and failure flag if present. These records are provided to the mutation operator as diagnostic context, while ranking depends only on $\widehat{F}_q(h)$ and deterministic tie-breaking keys.

Candidates are ordered by ascending empirical CVaR. For candidates h_a and h_b, h_a precedes h_b if $\widehat{F}_q(h_a) < \widehat{F}_q(h_b)$. If their CVaR scores match, the empirical mean $\bar{z}(h) = \frac{1}{m} \sum_{j=1}^{m} z_j$ is the first tie-breaking key; if means match, $\max_j z_j$ is used. Remaining ties are broken by a deterministic code-level key, such as creation order or source length. This procedure defines a total order over all scored candidates.

For an accumulated multiset $Z(h)$, the same computation is applied after sorting all stored losses. If $Z(h)$ contains m_h entries, then $r_h = \alpha m_h$ and $k_h = \lceil r_h \rceil$, and the score is computed from the largest k_h sorted entries with the corresponding fractional boundary weight. A candidate with only penalty losses receives $\widehat{F}_q(h) = \ell_{\text{fail}}$. A candidate with mixed successful and failed executions is scored by the sorted mixture, so failures enter the upper tail whenever their penalties are among the largest losses.

3.3 Program Mutation and Evolutionary Search

The search module maps scored programs to a new population. Its inputs at generation t are the current population $\mathcal{P}_t$, stored losses and scores for its members, and the mutation budget; its outputs are $\mathcal{P}_{t+1}$ and an updated archive containing the lowest-scoring program observed so far.

Each candidate is stored as source code implementing the required callable interface. The mutation operator receives the parent source code, interface signature, task specification, and optionally the parent's stored tail records. It returns a complete source-code candidate $\tilde{h}$. The returned text is parsed to extract a single program body, compiled

Method	Bin Packing		CVRP		Flow-Shop	
	Mean ↓	$\text{CVaR}_{0.90}$ ↓	Mean ↓	$\text{CVaR}_{0.90}$ ↓	Mean ↓	$\text{CVaR}_{0.90}$ ↓
Handcrafted heuristic	7.41±0.08	11.79±0.24	8.12±0.10	12.96±0.27	6.76±0.09	10.84±0.21
Direct LLM sampling	7.03±0.13	11.18±0.31	7.80±0.15	12.47±0.35	6.49±0.11	10.29±0.28
FunSearch-style	6.66±0.09	10.51±0.23	7.37±0.11	11.62±0.26	6.16±0.08	9.78±0.21
EoH-style	6.42±0.06	10.03±0.18	7.18±0.08	11.21±0.21	5.99±0.05	9.39±0.16
Mean-RMD	6.56±0.06	9.88±0.15	7.20±0.06	11.04±0.17	**5.95±0.04**	9.26±0.13
CVaR-RMD	**6.31±0.04**	**9.29±0.11**	**7.08±0.05**	**10.41±0.14**	5.96±0.04	**8.81±0.11**

Table 1: Held-out performance across 15 discovery seeds. Values are mean ± standard deviation. Lower is better.

Method	Mean loss ↓	$\text{CVaR}_{0.90}$ ↓	Max loss ↓	Failure rate (%) ↓
Mean-RMD	7.59±0.08	12.61±0.24	18.34±0.51	3.5±0.5
CVaR-RMD	**7.48±0.06**	**11.63±0.18**	**16.61±0.42**	**2.0±0.3**

Table 2: Transfer evaluation on larger held-out instances across 15 discovery seeds.

in the harness, and checked against the required callable signature. If validation fails, the offspring receives penalty losses for the current batch. If validation succeeds, the offspring is executed on sampled environments and scored by the empirical CVaR computation in Section 3.2.

Parent selection uses the current ranking. Let R_t be the ordered list from sorting $\mathcal{P}_t$. The parent pool is $\mathcal{A}_t = \{R_t[1], \ldots, R_t[M]\}$ with $M \leq N$. To produce one offspring, parent rank $a \in \{1, \ldots, M\}$ is sampled with probability

$$p_a = \frac{\exp(-\tau a)}{\sum_{b=1}^{M} \exp(-\tau b)}, \qquad \tau \geq 0,$$

where τ controls selection pressure toward higher-ranked programs. The selected parent $h_p = R_t[a]$ is mutated to produce $\tilde{h}$. Repeated draws are allowed, so multiple offspring may share a parent within one generation.

After K offspring are produced, the pool is $\mathcal{C}_t = \mathcal{P}_t \cup \mathcal{O}_t$, where $\mathcal{O}_t = \{\tilde{h}_t^{(1)}, \ldots, \tilde{h}_t^{(K)}\}$. Every member of $\mathcal{C}_t$ has an empirical CVaR score from execution or a penalty score from validation failure. The pool is sorted by the ranking rule, and

$$\mathcal{P}_{t+1} = \{C_t[1], \ldots, C_t[N]\},$$

where $C_t[a]$ is the a-th ranked candidate in $\mathcal{C}_t$. The archive stores the best program, or a bounded set of best programs, under the same ordering. After replacement, the archive is updated by comparing its contents with the highest-ranked member of $\mathcal{P}_{t+1}$.

The loop stops when the evaluation, generation, or execution-time budget is exhausted. The returned program is the lowest-scoring archived program under empirical CVaR and deterministic tie-breaking. At deployment, this program is executed directly through the same callable interface on a new environment input; no mutation step or language-model call is performed.

4 Experiments

4.1 Setup and Protocol

We evaluate whether tail-risk improves the robustness of LLM-discovered metaheuristics compared with mean-based selection. The benchmark suite covers online bin packing, capacitated vehicle routing (CVRP), and permutation flow-shop scheduling, exposing brittleness from online decisions, feasibility constraints, and sequencing interactions. CVaR-RMD and Mean-RMD use the same code interface, validator, mutation operator, population size, generation budget, archive mechanism, and held-out evaluation. They differ only in candidate ranking: empirical CVaR for CVaR-RMD and empirical mean loss for Mean-RMD.

Each method uses 15 independent discovery seeds, $\{0, 1, \ldots, 14\}$. After discovery, the best archived program from each seed is frozen and evaluated on shared held-out environments. We report the mean, standard deviation, and standardized effect sizes across discovery seeds. Significance is assessed by paired two-sided bootstrap tests with 10,000 resamples over archived programs and shared test environments. Unless noted otherwise, comparisons use Mean-RMD as the baseline.

The mutation model is GPT-4o with temperature 0.7, top-p 0.95, maximum output length 2048, and up to three retries for malformed code. Prompts contain the task definition, required function sig-

Setting	Mean loss ↓	$\mathrm{CVaR}_{0.90}$ ↓	Failure rate (%) ↓
Batch size 32	6.50±0.06	9.91±0.17	2.0±0.3
Batch size 64	**6.45±0.04**	9.50±0.11	1.4±0.2
Batch size 128	6.43±0.04	**9.42±0.10**	**1.3±0.2**
Failure penalty $\ell_{\mathrm{fail}} = 20$	6.44±0.05	9.63±0.13	1.7±0.3
Failure penalty $\ell_{\mathrm{fail}} = 30$	6.47±0.05	9.51±0.12	**1.3±0.2**

Table 3: Sensitivity analysis across tasks using 15 discovery seeds.

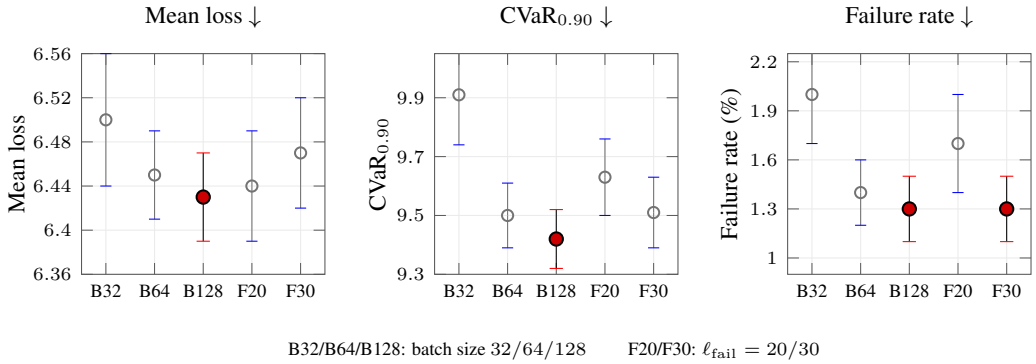


Figure 3: Sensitivity analysis across tasks using 15 discovery seeds. Error bars denote standard deviations; filled markers indicate the best value for each metric, including tied best values.

nature, parent program, validation constraints, and a compact summary of its high-loss cases. The model must return a complete Python function preserving the interface. Generated code runs in a sandbox with fixed package versions. LLM calls occur only during discovery.

Training uses medium-scale instances, whereas held-out testing covers in-distribution and larger transfer instances. Bin packing uses uniform, truncated-normal, and heavy-tailed beta item distributions, with 100–300 discovery items and 400–800 transfer items. CVRP uses uniform and clustered Euclidean generators, with 50–100 discovery customers and 150–300 transfer customers. Flow-shop scheduling uses uniform, correlated, and bottleneck processing-time families, with 20–50 discovery jobs and 80–120 transfer jobs.

At each generation, all candidates are evaluated on a shared batch of 64 environments. Losses accumulate across generations until archived candidates have at least 512 evaluations, stabilizing tail estimates. The population has $N = 24$ programs, with $M = 8$ parents eligible for mutation and $K = 16$ offspring per generation. Each discovery run lasts 40 generations and yields 640 LLM-generated offspring per seed. The default tail level is $q = 0.90$. Execution is capped at 2 seconds per training instance and 10 seconds per transfer instance. Invalid code, interface violations, infeasible outputs, exceptions, and timeouts receive $\ell_{\mathrm{fail}} = 25$.

Losses are normalized by task-specific references: the continuous bin-capacity lower bound for bin packing, Clarke–Wright savings with deterministic 2-opt for CVRP, and NEH for flow-shop scheduling. Lower values are better. We report mean loss, empirical $\mathrm{CVaR}_{0.90}$, maximum loss, and failure rate. The Appendix provides per-task breakdowns and sensitivity analyses.

4.2 Baselines

We compare four baseline groups. Handcrafted heuristics comprise first-fit and best-fit for bin packing, Clarke–Wright savings and cheapest insertion for CVRP, and NEH with shortest-processing-time dispatching for flow-shop scheduling. Direct LLM sampling draws as many candidate programs as one discovery run without evolutionary replacement. FunSearch-style (Romera-Paredes et al., 2024) and EoH-style (Liu et al., 2024a) variants use the same infrastructure, generators, parsers, validators, budgets, and LLM settings as CVaR-RMD but retain mean-performance selection. Mean-RMD is the closest control, using the same evolutionary loop as CVaR-RMD but ranking candidates by empirical mean loss.

4.3 Held-Out Robustness

Table 1 shows that CVaR-RMD achieves the lowest $\mathrm{CVaR}_{0.90}$ on all three tasks. Relative to Mean-

Method	Mean loss ↓	$\mathrm{CVaR}_{0.90}$ ↓	Max loss ↓	Failure rate ↓
Mean-RMD	6.47	10.03	15.72	2.3%
Mean+Std-RMD	6.44	9.70	15.12	1.7%
Quantile-RMD	6.49	9.71	14.98	1.6%
Hard-Mining-RMD	6.46	9.66	14.81	1.5%
Max-RMD	6.61	9.62	**14.32**	**1.3%**
CVaR-RMD	**6.45**	**9.50**	14.55	1.4%

Table 4: Comparison with alternative risk-aware selection objectives. Values are equally weighted macro-averages across bin packing, CVRP, and flow-shop scheduling. All methods use matched candidate-generation and program-execution budgets. Lower is better.

External benchmark	Mean-RMD	CVaR-RMD	Relative change
OR-Library bin packing	12.41	**11.63**	-6.3%
CVRPLIB	14.62	**13.31**	-9.0%
Taillard flow shop	12.78	**11.94**	-6.6%

Table 5: Transfer performance on independently curated public benchmark instances. The metric is $\mathrm{CVaR}_{0.90}$, and lower values are better. Programs are frozen before external evaluation.

RMD, it reduces tail loss by 6.0% on bin packing, 5.7% on CVRP, and 4.9% on flow-shop scheduling. Paired bootstrap tests yield $p < 0.01$ for all tasks, with Cohen's d values of 1.21, 1.08, and 0.94, respectively. These gains therefore represent a consistent shift in the discovered-program distribution rather than isolated favorable runs.

Mean-loss differences are smaller, as expected from the objective. CVaR-RMD improves mean loss by 3.8% on bin packing and 1.7% on CVRP, while leaving flow-shop mean loss statistically unchanged. Risk-sensitive selection thus mainly suppresses high-loss outcomes without sacrificing average performance. Direct LLM sampling underperforms all evolutionary variants, confirming the value of iterative evaluation and replacement. Among discovery methods, CVaR-RMD provides the strongest upper-tail robustness while remaining competitive in mean loss.

4.4 Transfer to Larger Instances

We freeze each discovered program and evaluate it on larger unseen instances with new random seeds. Instance families remain unchanged, but their scale parameters exceed the discovery range.

Table 2 shows that the robustness advantage persists under scale transfer. Relative to Mean-RMD, CVaR-RMD reduces transfer $\mathrm{CVaR}_{0.90}$ by 7.8%, maximum loss by 9.4%, and failure rate by 1.5 percentage points, with all differences significant at $p < 0.02$. The strongest effect occurs on CVRP, where mean-based discovery sometimes produces route-construction logic that becomes infeasible for larger customer sets. Flow-shop scheduling shows smaller average gains but still benefits from lower maximum loss.

4.5 Comparison with Alternative Risk-Aware Objectives

Mean-RMD controls the comparison between expectation-based and tail-risk selection but does not show whether CVaR is preferable to other robustness objectives. We therefore evaluate four additional risk-aware variants using the same LLM mutation model, population size, search budget, validation procedure, and program–environment execution budget as CVaR-RMD.

MEAN+STD-RMD ranks candidates by the sum of empirical mean loss and standard deviation. QUANTILE-RMD uses empirical $\mathrm{VaR}_{0.90}$, while MAX-RMD uses maximum observed loss. Finally, HARD-MINING-RMD increases the sampling probability of environments that previously produced high losses. These controls test whether the robustness gain is specific to CVaR or reflects risk-sensitive selection more broadly.

Table 4 reports equally weighted macro-averages across the three tasks. All methods use matched search and evaluation budgets. CVaR-RMD achieves the lowest $\mathrm{CVaR}_{0.90}$ while remaining competitive in mean loss. Max-RMD achieves the lowest maximum loss and failure rate but substantially higher mean loss. The other risk-aware objectives also improve tail performance over Mean-RMD.

These results show that CVaR is not the only ef-

Variant	Mean loss ↓	$\mathrm{CVaR}_{0.90}$ ↓	Failure rate (%) ↓
Full CVaR-RMD	**6.45±0.04**	**9.50±0.11**	**1.4±0.2**
Mean ranking	6.47±0.05	10.06±0.15	2.3±0.3
No accumulated losses	6.53±0.06	10.22±0.17	2.7±0.4
No tail records in prompt	6.48±0.05	9.81±0.14	1.9±0.3
Tail level $q = 0.80$	6.48±0.05	9.87±0.14	1.8±0.3
Tail level $q = 0.95$	6.57±0.06	9.56±0.13	1.5±0.2

Table 6: Ablation results averaged over tasks across 15 discovery seeds.

fective robustness objective. Max-RMD suppresses the worst observed outcome more aggressively and achieves the lowest failure rate, but with worse mean performance. Under the evaluated protocol, CVaR-RMD provides the best observed mean–tail trade-off. We therefore do not claim that CVaR universally dominates other risk measures. Instead, the results establish CVaR as an effective risk-sensitive alternative when average performance and upper-tail robustness both matter.

4.6 Transfer to Independently Curated Public Benchmarks

To reduce dependence on synthetic discovery generators, we also evaluate the frozen programs on independently curated public benchmark instances. None are used for program discovery, prompt construction, hyperparameter selection, or archive selection.

The external evaluation uses OR-Library instances for bin packing, CVRPLIB instances for capacitated vehicle routing, and Taillard instances for permutation flow-shop scheduling. We apply the same programs frozen after the original discovery runs and report their $\mathrm{CVaR}_{0.90}$ on these instances.

Table 5 shows that CVaR-RMD reduces $\mathrm{CVaR}_{0.90}$ on all three external benchmark collections: by 6.3% on OR-Library bin packing, 9.0% on CVRPLIB, and 6.6% on Taillard flow-shop instances. The largest reduction occurs on CVRPLIB, whose independently curated instances expose routing and capacity patterns differing from the synthetic discovery instances.

These results support within-domain transfer to independently curated benchmarks. They do not establish cross-domain transfer, robustness to arbitrary distribution shifts, or operational deployment performance. We therefore restrict the generalization claim to improved upper-tail robustness on unseen scales, instance seeds, and independent within-domain benchmarks under the evaluated protocol.

4.7 Ablations and Sensitivity

We ablate components distinguishing CVaR-RMD from the mean-based control while fixing prompts, validators, budgets, populations, and held-out tests.

Table 6 identifies CVaR ranking as the main improvement driver. Replacing it with mean ranking increases $\mathrm{CVaR}_{0.90}$ by 5.9% and raises failure rate from 1.4% to 2.3%. Removing accumulated losses performs worse because fewer evaluations yield noisier tail estimates. Prompted tail-loss summaries provide a smaller additional benefit by informing mutations about adverse execution modes. Tail-level comparisons reveal the expected tradeoff: $q = 0.80$ offers less protection against rare failures, whereas $q = 0.95$ suppresses them more aggressively but weakens mean performance. The default $q = 0.90$ provides the best observed balance.

Table 3 and Figure 3 show that CVaR-RMD remains stable under moderate changes in evaluation batch size and failure penalty. Smaller batches increase tail-estimation variance, whereas larger batches improve robustness at higher evaluation cost. Increasing the failure penalty further discourages infeasible programs but mildly reduces search flexibility in mean loss. These results support the default configuration as a practical robustness-cost compromise.

5 Conclusion

We presented CVaR-RMD, a risk-sensitive framework that replaces mean-based fitness with CVaR selection in LLM-driven metaheuristic discovery. Across three combinatorial optimization tasks, CVaR-RMD improved upper-tail performance and reduced failure rates while maintaining competitive mean performance. These benefits persisted under within-domain transfer to larger unseen instances and independently curated benchmarks. The results show that tail-risk-based selection can produce more reliable heuristics under benchmark distribution shifts without adding architectural components to the discovery pipeline.

Limitations

The empirical evaluation focuses on three representative combinatorial optimization tasks, and further validation on additional problem classes would help assess the generality of the approach. CVaR-RMD also introduces a robustness hyperparameter through the tail level q, and while a single default value performs consistently well in the experiments, the optimal choice may vary across applications depending on the desired trade-off between average and worst-case performance.

LLM Usage Statement

During the preparation of this manuscript, large language models (LLM) were used to improve the grammar, wording, and overall readability of the text. They were not used to generate scientific content, conduct data analysis, or draw conclusions. All LLM-assisted revisions were carefully reviewed and approved by the authors.

Acknowledgements

The work was supported by the National Natural Science Foundation of China (Grant No. 62471287)

References

Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. 2009. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Principles and Practice of Constraint Programming – CP 2009*, volume 5732 of *Lecture Notes in Computer Science*, pages 142–157. Springer.

Philippe Artzner, Freddy Delbaen, Jean-Marc Eber, and David Heath. 1999. Coherent measures of risk. *Mathematical Finance*, 9(3):203–228.

Thomas Bäck. 1996. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press.

Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. 2017. Neural combinatorial optimization with reinforcement learning. In *Proceedings of the International Conference on Learning Representations Workshop*.

Aharon Ben-Tal, Laurent El Ghaoui, and Arkadi Nemirovski. 2009. *Robust Optimization*. Princeton University Press.

Dimitris Bertsimas, David B. Brown, and Constantine Caramanis. 2011. Theory and applications of robust optimization. *SIAM Review*, 53(3):464–501.

Hans-Georg Beyer and Bernhard Sendhoff. 2007. Robust optimization: A comprehensive survey. *Computer Methods in Applied Mechanics and Engineering*, 196(33–34):3190–3218.

Christian Blum and Andrea Roli. 2003. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308.

Jürgen Branke, Kalyanmoy Deb, Kaisa Miettinen, and Roman Słowiński, editors. 2008. *Multiobjective Optimization: Interactive and Evolutionary Approaches*, volume 5252 of *Lecture Notes in Computer Science*. Springer.

Edmund K. Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. 2013. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724.

Yinlam Chow, Aviv Tamar, Shie Mannor, and Marco Pavone. 2015. Risk-sensitive and robust decision-making: A CVaR optimization approach. In *Advances in Neural Information Processing Systems*, volume 28, pages 1522–1530.

Peter Cowling, Graham Kendall, and Eric Soubeiga. 2001. A hyperheuristic approach to scheduling a sales summit. In *Practice and Theory of Automated Timetabling III*, volume 2079 of *Lecture Notes in Computer Science*, pages 176–190. Springer.

Erick Delage and Yinyu Ye. 2010. Distributionally robust optimization under moment uncertainty with application to data-driven problems. *Operations Research*, 58(3):595–612.

John C. Duchi and Hongseok Namkoong. 2021. Learning models with uniform performance via distributionally robust optimization. *The Annals of Statistics*, 49(3):1378–1406.

A. E. Eiben and J. E. Smith. 2003. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer.

David E. Goldberg and Kalyanmoy Deb. 1991. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms*, volume 1, pages 69–93. Morgan Kaufmann.

Holger H. Hoos and Thomas Stützle. 2004. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann.

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization*, Lecture Notes in Computer Science, pages 507–523. Springer.

Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. 2009. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306.

Yaochu Jin and Jürgen Branke. 2005. Evolutionary optimization in uncertain environments: A survey. *IEEE Transactions on Evolutionary Computation*, 9(3):303–317.

David S. Johnson and Lyle A. McGeoch. 1997. The traveling salesman problem: A case study in local optimization. In Emile H. L. Aarts and Jan Karel Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. Wiley.

Chaitanya K. Joshi, Thomas Laurent, and Xavier Bresson. 2019. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint arXiv:1906.01227*.

Wouter Kool, Herke van Hoof, and Max Welling. 2019. Attention, learn to solve routing problems! In *Proceedings of the International Conference on Learning Representations*.

John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

William B. Langdon and Riccardo Poli. 2002. *Foundations of Genetic Programming*. Springer.

Fei Liu, Xialiang Tong, 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 *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 32201–32223. PMLR.

Tennison Liu, Nicolás Astorga, Nabeel Seedat, and Mihaela van der Schaar. 2024b. Large language models to enhance bayesian optimization. In *Proceedings of the International Conference on Learning Representations*.

Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Thomas Stützle, and Mauro Birattari. 2016. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58.

Gabriela Ochoa, Matthew Hyde, Tim Curtois, Jose A. Vazquez-Rodriguez, James Walker, Michel Gendreau, Graham Kendall, Barry McCollum, Andrew J. Parkes, Sanja Petrovic, and Edmund K. Burke. 2012. HyFlex: A benchmark framework for cross-domain heuristic search. In *Evolutionary Computation in Combinatorial Optimization*, volume 7245 of *Lecture Notes in Computer Science*, pages 136–147. Springer.

Michael O'Neill and Conor Ryan. 2001. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358.

Georg Ch. Pflug. 2000. Some remarks on the value-at-risk and the conditional value-at-risk. In Stanislav P. Uryasev, editor, *Probabilistic Constrained Optimization: Methodology and Applications*, pages 272–281. Kluwer Academic Publishers.

Hamed Rahimian and Sanjay Mehrotra. 2019. Distributionally robust optimization: A review. *arXiv preprint arXiv:1908.05659*.

Gerhard Reinelt. 1991. TSPLIB – a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384.

John R. Rice. 1976. The algorithm selection problem. *Advances in Computers*, 15:65–118.

R. Tyrrell Rockafellar and Stanislav Uryasev. 2000. Optimization of conditional value-at-risk. *Journal of Risk*, 2(3):21–42.

R. Tyrrell Rockafellar and Stanislav Uryasev. 2002. Conditional value-at-risk for general loss distributions. *Journal of Banking and Finance*, 26(7):1443–1471.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. 2024. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475.

Alexander Shapiro, Darinka Dentcheva, and Andrzej Ruszczyński. 2009. *Lectures on Stochastic Programming: Modeling and Theory*. SIAM.

El-Ghazali Talbi. 2009. *Metaheuristics: From Design to Implementation*. Wiley Series on Parallel and Distributed Computing. Wiley.

Aviv Tamar, Yonatan Glassner, and Shie Mannor. 2015. Optimizing the CVaR via sampling. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29.

Niki van Stein and Thomas Bäck. 2024. LLaMEA: A large language model evolutionary algorithm for automatically generating metaheuristics. *IEEE Transactions on Evolutionary Computation*.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *Advances in Neural Information Processing Systems*, volume 28.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. 2024. ReEvo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems*, volume 37.

Appendix

This appendix provides supplementary material for CVaR-RMD. Appendix A describes the three benchmark tasks (bin packing, CVRP, flow-shop scheduling), instance distributions, and the held-out evaluation protocol across 15 discovery seeds. Appendix B details the empirical CVaR computation, population management, archive mechanism, and the shared code interface with Mean-RMD. Appendix C reports full tail-risk and mean-performance tables, sensitivity analyses on the CVaR confidence level, and per-seed variance statistics. Appendix D provides the prompt templates used for candidate generation and mutation in the LLM-driven discovery loop. Appendix E illustrates specific discovered metaheuristics where CVaR selection avoids brittle solutions that score well on average but fail on tail instances.

A Environment Details

A.1 Datasets and Task Settings

We evaluate LLM-discovered metaheuristics on three combinatorial optimization benchmarks: online bin packing, capacitated vehicle routing, and permutation flow-shop scheduling. These tasks are selected to cover online allocation decisions, hard feasibility constraints, and long-horizon sequencing interactions.

For online bin packing, instances consist of sequentially arriving items that must be assigned to bins of fixed capacity. Discovery uses item sequences with 100–300 items drawn from uniform, truncated-normal, and heavy-tailed beta distributions, while transfer evaluation uses larger sequences with 400–800 items.

For CVRP, each instance contains Euclidean customer locations, customer demands, a depot, and a vehicle-capacity constraint. Discovery uses uniform and clustered generators with 50–100 customers, and held-out transfer evaluation uses 150–300 customers generated with unseen random seeds.

For flow-shop scheduling, each instance specifies processing times for a set of jobs across machines, and the objective is to construct a job permutation with low makespan. Discovery uses 20–50 jobs from uniform, correlated, and bottleneck processing-time families, and transfer evaluation uses larger instances with 80–120 jobs.

All tasks are formulated as program-synthesis problems in which the LLM proposes a complete Python heuristic satisfying a fixed function signature. Prompts include the task definition, interface requirements, validation constraints, the parent program, and a compact summary of high-loss cases. No held-out test instances are used for discovery or prompt construction. Hyperparameters are fixed before final evaluation, and the same discovery and evaluation splits are used for CVaR-RMD, Mean-RMD, and all implemented baselines.

A.2 Preprocessing and Evaluation

Generated programs are parsed as Python functions and checked against the required interface before execution. We reject malformed code, missing functions, invalid return types, infeasible solutions, runtime exceptions, and timeouts. Such cases receive the fixed failure loss $\ell_{\text{fail}} = 25$. Candidate execution is capped at 2 seconds per training instance and 10 seconds per transfer instance.

Losses are normalized by task-specific reference procedures to make results comparable across instance families and scales. Bin-packing losses are normalized by the continuous bin-capacity lower bound, CVRP losses by Clarke–Wright savings with deterministic 2-opt, and flow-shop losses by the NEH heuristic. Lower values indicate better performance.

At each generation, candidates are evaluated on a shared batch of 64 environments. Loss histories are accumulated across generations until archived candidates have at least 512 evaluations, which is used to stabilize empirical tail estimates. CVaR-RMD ranks candidates by empirical CVaR_q with default tail level $q = 0.90$, while Mean-RMD ranks candidates by empirical mean loss. For a loss sample $\{\ell_i\}_{i=1}^n$, empirical $\mathrm{CVaR}_{0.90}$ is computed as the average loss over the largest 10% of observed losses.

Each method is run with 15 independent discovery seeds, $\{0, 1, \ldots, 14\}$. After discovery, the best archived program from each seed is frozen and evaluated on common held-out environments. We report mean loss, empirical $\mathrm{CVaR}_{0.90}$, maximum loss, and failure rate. Aggregate results are reported as mean, standard deviation, and standardized effect size across discovery seeds. Statistical significance is assessed using paired two-sided bootstrap tests with 10,000 resamples over archived programs and shared test environments.

Baselines include handcrafted heuristics, direct LLM sampling, FunSearch-style discovery, EoH-style discovery, and Mean-RMD. All discovery

baselines use the same generators, parsers, validators, mutation budget, population size, archive mechanism, and held-out evaluation protocol.

B Implementation Details

All methods are implemented in a common Python infrastructure with fixed package versions and sandboxed execution for generated programs. CVaR-RMD and Mean-RMD share the same evolutionary loop and differ only in the candidate-ranking statistic. The population contains $N = 24$ programs, $M = 8$ parents are eligible for mutation at each generation, and each generation produces $K = 16$ offspring. Each discovery run lasts 40 generations, yielding 640 LLM-generated offspring per seed and at most 664 evaluated candidate programs per run.

The mutation model is GPT-4o. We use temperature 0.7, top-p 0.95, maximum output length 2048, and up to three retries for malformed code. The model is used only during discovery. Final held-out evaluation executes frozen programs without additional LLM calls.

The default CVaR-RMD configuration uses tail level $q = 0.90$, batch size 64, accumulated loss histories, tail-case summaries in prompts, and failure penalty $\ell_{\text{fail}} = 25$. Sensitivity experiments vary the batch size, tail level, and failure penalty while keeping all other components fixed.

Experiments are run on an AMD EPYC 7543 workstation with 256GB RAM. Average wall-clock times for CVaR-RMD are 5.9 hours for bin packing, 8.6 hours for CVRP, and 7.3 hours for flow-shop scheduling. Mean-RMD uses the same number of LLM calls, generated offspring, validation checks, and candidate executions; its runtime is within 4% of CVaR-RMD. The additional cost of CVaR-RMD is limited to sorting accumulated loss vectors for empirical tail-risk estimation.

C More Experiment Results

C.1 Per-Task Transfer Results

Table 7 decomposes the large-instance transfer results by task. CVaR-RMD consistently reduces tail loss and maximum loss across all three benchmarks. The largest relative reduction in $\mathrm{CVaR}_{0.90}$ appears on CVRP, where larger customer sets expose more feasibility and route-construction failures under mean-based selection.

C.2 Robustness Across Instance Families

Table 8 reports held-out performance by instance family. The advantage of CVaR-RMD is more pronounced on heavy-tailed, clustered, and bottleneck families, which contain more adverse cases than the corresponding uniform generators. This supports the interpretation that CVaR-based selection primarily improves robustness under difficult distributional conditions.

C.3 Alternative Tail Levels at Evaluation Time

Table 9 evaluates the frozen programs with different tail levels. Although CVaR-RMD is selected using $q = 0.90$, its advantage remains visible at both milder and more extreme tail levels. The improvement is largest at $q = 0.95$, indicating that the selected programs reduce severe adverse outcomes rather than only optimizing the specific training quantile.

C.4 Discovery Progress

Table 10 reports the held-out performance of the best archived program selected after different discovery budgets. CVaR-RMD improves steadily with more generations and remains better than Mean-RMD in tail loss throughout the later stages of discovery. Mean loss converges more closely, consistent with the main observation that the primary benefit of CVaR-RMD is reduced upper-tail risk.

C.5 Run-to-Run Stability

Table 11 reports dispersion statistics over discovery seeds. CVaR-RMD reduces the interquartile range and worst-seed loss on all tasks, showing that the improvement is not only due to better average seed performance. The reduction in worst-seed $\mathrm{CVaR}_{0.90}$ is particularly clear on CVRP.

C.6 Statistical Comparison

Table 12 provides additional paired comparisons between CVaR-RMD and Mean-RMD. The effect sizes are larger for $\mathrm{CVaR}_{0.90}$ than for mean loss, matching the intended behavior of the risk-sensitive objective.

D Prompt Templates

D.1 Overview

This appendix documents the prompt templates used for LLM-guided program discovery. The

Task	Method	Mean loss ↓	$\text{CVaR}_{0.90}$ ↓	Max loss ↓	Failure rate (%) ↓
Bin Packing	Mean-RMD	7.18±0.09	11.82±0.27	17.41±0.55	2.8±0.5
	CVaR-RMD	**7.08±0.07**	**10.96±0.20**	**15.92±0.44**	**1.7±0.3**
CVRP	Mean-RMD	8.34±0.11	13.91±0.31	20.46±0.68	4.9±0.7
	CVaR-RMD	**8.18±0.09**	**12.69±0.24**	**18.12±0.53**	**2.8±0.5**
Flow-Shop	Mean-RMD	7.25±0.08	12.10±0.24	17.16±0.49	2.7±0.4
	CVaR-RMD	**7.19±0.07**	**11.25±0.19**	**15.78±0.41**	**1.6±0.3**

Table 7: Per-task transfer results on larger held-out instances across 15 discovery seeds. Values are mean ± standard deviation.

Task	Inst. family	Method	Mean loss ↓	$\text{CVaR}_{0.90}$ ↓	Failure rate (%) ↓
Bin Packing	Uniform	Mean-RMD	5.94±0.05	8.84±0.14	1.3±0.2
		CVaR-RMD	**5.90±0.04**	**8.42±0.11**	**0.9±0.2**
	Truncated normal	Mean-RMD	6.21±0.06	9.47±0.16	1.7±0.3
		CVaR-RMD	**6.15±0.05**	**8.89±0.13**	**1.1±0.2**
	Heavy-tailed beta	Mean-RMD	6.92±0.08	11.33±0.22	2.9±0.4
		CVaR-RMD	**6.87±0.07**	**10.55±0.17**	**1.8±0.3**
CVRP	Uniform	Mean-RMD	6.82±0.07	10.38±0.17	1.8±0.3
		CVaR-RMD	**6.77±0.06**	**9.84±0.14**	**1.2±0.2**
	Clustered	Mean-RMD	7.46±0.08	11.70±0.22	3.1±0.4
		CVaR-RMD	**7.38±0.07**	**10.94±0.18**	**1.9±0.3**
Flow-Shop	Uniform	Mean-RMD	**5.67±0.04**	8.62±0.12	1.0±0.2
		CVaR-RMD	5.68±0.04	**8.28±0.10**	**0.7±0.1**
	Correlated	Mean-RMD	**5.89±0.05**	9.14±0.14	1.3±0.2
		CVaR-RMD	5.90±0.04	**8.71±0.12**	**0.9±0.2**
	Bottleneck	Mean-RMD	6.29±0.06	10.02±0.18	2.2±0.3
		CVaR-RMD	**6.27±0.05**	**9.44±0.15**	**1.5±0.2**

Table 8: Held-out robustness by instance family across 15 discovery seeds.

prompts are used only during discovery to generate or mutate executable heuristic programs. After discovery, archived programs are frozen and evaluated directly through the prescribed callable interface without further LLM calls. All prompts require the model to return complete Python source code satisfying the task interface; validation, feasibility checking, loss computation, and model comparison are performed by deterministic code rather than by an LLM judge.

D.2 Notation

We use the following notation in prompt templates.

- $\mathcal{T}$ denotes the optimization task, one of online bin packing, capacitated vehicle routing, or permutation flow-shop scheduling.
- `{TASK_SPECIFICATION}` is a natural-language specification of $\mathcal{T}$, including the objective, feasibility constraints, input format, and output format.
- `{FUNCTION_SIGNATURE}` is the required Python callable signature. Every generated program must implement exactly this interface.
- `{VALIDATION_CONSTRAINTS}` lists syntactic, interface, runtime, and feasibility requirements enforced by the sandbox validator.
- `{PARENT_PROGRAM}` is the complete source code of the selected parent heuristic.
- `{PARENT_SCORE}` is the scalar fitness value used for ranking the parent. In CVaR-RMD

Method	Bin Packing		CVRP		Flow-Shop	
	$\mathrm{CVaR}_{0.80}\downarrow$	$\mathrm{CVaR}_{0.95}\downarrow$	$\mathrm{CVaR}_{0.80}\downarrow$	$\mathrm{CVaR}_{0.95}\downarrow$	$\mathrm{CVaR}_{0.80}\downarrow$	$\mathrm{CVaR}_{0.95}\downarrow$
Mean-RMD	8.71±0.12	10.87±0.19	9.74±0.14	12.29±0.23	8.18±0.11	10.21±0.17
CVaR-RMD	**8.29±0.09**	**10.06±0.15**	**9.25±0.12**	**11.47±0.20**	**7.84±0.09**	**9.61±0.14**

Table 9: Evaluation with alternative CVaR tail levels across 15 discovery seeds. Programs are the same frozen programs used in the main held-out evaluation.

Generation	Mean-RMD			CVaR-RMD		
	Mean loss ↓	$\mathrm{CVaR}_{0.90}\downarrow$	Failure rate (%) ↓	Mean loss ↓	$\mathrm{CVaR}_{0.90}\downarrow$	Failure rate (%) ↓
10	6.78±0.08	10.71±0.19	3.0±0.4	**6.72±0.07**	**10.29±0.17**	**2.5±0.4**
20	6.59±0.06	10.32±0.17	2.6±0.4	**6.54±0.06**	**9.83±0.14**	**1.9±0.3**
30	6.51±0.05	10.12±0.16	2.4±0.3	**6.46±0.05**	**9.60±0.12**	**1.6±0.3**
40	6.47±0.05	10.03±0.15	2.3±0.3	**6.42±0.04**	**9.48±0.11**	**1.4±0.2**

Table 10: Discovery progress averaged across tasks and 15 discovery seeds.

this is empirical CVaR_q; in mean-based variants it is empirical mean loss.

- `{MEAN_LOSS}`, `{CVAR_LOSS}`, `{MAX_LOSS}`, and `{FAILURE_RATE}` are deterministic summaries computed from the parent's stored losses.
- `{TAIL_LEVEL}` is the CVaR tail level q.
- `{TAIL_RECORDS}` is a compact deterministic summary of high-loss environments for the parent, including generator identity, scale value, instance seed, execution seed, realized loss, and failure flag.
- `{REFERENCE_HEURISTICS}` optionally contains names of classical heuristics used only as task context, such as first-fit decreasing, best-fit, Clarke–Wright savings, cheapest insertion, NEH, or shortest-processing-time dispatching.
- `{ARCHIVE_PROGRAMS}` denotes previously discovered high-performing programs when a baseline variant exposes archive exemplars.

D.3 Main Inference Prompt

Purpose. This prompt implements the mutation operator in CVaR-RMD. It receives a parent program, the required task interface, validation constraints, and high-loss diagnostic records. It asks the model to produce a complete mutated Python function that preserves the interface while improving robustness on adverse cases.

Inputs. `{TASK_SPECIFICATION}`, `{FUNCTION_SIGNATURE}`, `{VALIDATION_CONSTRAINTS}`, `{PARENT_PROGRAM}`, `{PARENT_SCORE}`, `{MEAN_LOSS}`, `{CVAR_LOSS}`, `{MAX_LOSS}`, `{FAILURE_RATE}`, `{TAIL_LEVEL}`, and `{TAIL_RECORDS}`.

Output. A single complete Python function implementing `{FUNCTION_SIGNATURE}`. The output must contain code only, with no Markdown fences, no explanatory text, and no additional top-level functions unless they are nested inside the required function.

D.4 Baseline Prompts

Purpose. Direct LLM sampling uses this prompt to generate candidate programs without evolutionary replacement. The same task interface, validation constraints, model settings, and output requirements are used as in CVaR-RMD, but no parent program or tail records are provided.

Inputs. `{TASK_SPECIFICATION}`, `{FUNCTION_SIGNATURE}`, `{VALIDATION_CONSTRAINTS}`, and optionally `{REFERENCE_HEURISTICS}`.

Output. A single complete Python function implementing `{FUNCTION_SIGNATURE}`.

Purpose. Mean-RMD uses the same mutation prompt as CVaR-RMD, because the two methods share the code interface, validator, mutation operator, population size, generation budget, archive mechanism, and held-out evaluation. The only algorithmic difference is the ranking score used for

Task	Method	Median $\mathrm{CVaR}_{0.90}\downarrow$	IQR $\downarrow$	Worst-seed $\mathrm{CVaR}_{0.90}\downarrow$	Best-seed $\mathrm{CVaR}_{0.90}\downarrow$
Bin Packing	Mean-RMD	9.86	0.21	10.17	9.61
	CVaR-RMD	**9.28**	**0.15**	**9.52**	**9.09**
CVRP	Mean-RMD	11.02	0.25	11.41	10.76
	CVaR-RMD	**10.40**	**0.19**	**10.72**	**10.18**
Flow-Shop	Mean-RMD	9.25	0.18	9.52	9.06
	CVaR-RMD	**8.80**	**0.15**	**9.05**	**8.63**

Table 11: Run-to-run stability of held-out $\mathrm{CVaR}_{0.90}$ across 15 discovery seeds.

Metric	Task	Mean-RMD	CVaR-RMD	Relative change	Cohen's d
Mean loss	Bin Packing	6.56 ± 0.06	$\mathbf{6.31\pm0.04}$	-3.8%	0.58
	CVRP	7.20 ± 0.06	$\mathbf{7.08\pm0.05}$	-1.7%	0.64
	Flow-Shop	$\mathbf{5.95\pm0.04}$	5.96 ± 0.04	$+0.2\%$	-0.18
$\mathrm{CVaR}_{0.90}$	Bin Packing	9.88 ± 0.15	$\mathbf{9.29\pm0.11}$	-6.0%	1.21
	CVRP	11.04 ± 0.17	$\mathbf{10.41\pm0.14}$	-5.7%	1.08
	Flow-Shop	9.26 ± 0.13	$\mathbf{8.81\pm0.11}$	-4.9%	0.94
Failure rate	Bin Packing	1.9 ± 0.3	$\mathbf{1.1\pm0.2}$	-0.8 pp	0.82
	CVRP	2.8 ± 0.4	$\mathbf{1.7\pm0.3}$	-1.1 pp	0.87
	Flow-Shop	2.1 ± 0.3	$\mathbf{1.3\pm0.2}$	-0.8 pp	0.76

Table 12: Additional statistical comparison between Mean-RMD and CVaR-RMD across 15 discovery seeds. Relative change is computed with respect to Mean-RMD; pp denotes percentage points.

selection. In the prompt, `{PARENT_SCORE}` is populated with empirical mean loss rather than empirical CVaR.

Inputs. The same inputs as Figure 4, with `{PARENT_SCORE}` equal to empirical mean loss.

Output. A single complete Python function implementing `{FUNCTION_SIGNATURE}`.

Purpose. FunSearch-style and EoH-style baselines are implemented in the same infrastructure and use mean-performance selection. When archive exemplars are exposed, the prompt asks the model to synthesize a new complete program from high-performing previously discovered programs while preserving the same callable interface.

Inputs. `{TASK_SPECIFICATION}`, `{FUNCTION_SIGNATURE}`, `{VALIDATION_CONSTRAINTS}`, `{ARCHIVE_PROGRAMS}`, `{MEAN_LOSS}`, `{MAX_LOSS}`, and `{FAILURE_RATE}`.

Output. A single complete Python function implementing `{FUNCTION_SIGNATURE}`.

D.5 Few-shot / Demonstration Prompts

No few-shot natural-language demonstrations are used in the default CVaR-RMD prompt. Candidate source programs may appear as parent programs or archive programs in mutation prompts, but they are executable program inputs rather than labeled few-shot examples. Therefore, the discovery protocol does not include a separate few-shot prompt template.

D.6 Reasoning or Rationale Prompts

No separate chain-of-thought, rationale-generation, or self-explanation prompt is used. The model is instructed to return code only. Diagnostic information from high-loss environments is computed deterministically by the evaluation harness and inserted into the mutation prompt as `{TAIL_RECORDS}`.

D.7 Data Generation or Pseudo-labeling Prompts

No LLM prompt is used for data generation or pseudo-labeling. Training and held-out environments are generated by deterministic benchmark generators from stored generator identities, scale values, instance seeds, and execution seeds. Feasi-

```
You are an expert Python programmer designing robust heuristic algorithms for combinatorial
optimization.

Task specification:
{TASK_SPECIFICATION}

Required function signature:
{FUNCTION_SIGNATURE}

Validation and execution constraints:
{VALIDATION_CONSTRAINTS}

The following parent program is a working heuristic candidate:
{PARENT_PROGRAM}

Parent performance summary:
fitness score: {PARENT_SCORE}
mean loss: {MEAN_LOSS}
CVaR@{TAIL_LEVEL}: {CVAR_LOSS}
maximum loss: {MAX_LOSS}
failure rate: {FAILURE_RATE}

High-loss environments observed for this parent:
{TAIL_RECORDS}

Write a mutated version of the parent program. Preserve the required function signature
exactly. The new program should remain simple, deterministic given its inputs and seed, and
robust on the high-loss cases above. Avoid changes that risk infeasible outputs, exceptions,
excessive runtime, or interface violations.

Return only the complete Python function. Do not include Markdown formatting, comments
outside the function, explanations, tests, or example usage.
```

Figure 4: Main CVaR-RMD mutation prompt.

bility labels, losses, failure flags, and tail records are produced by the sandbox validator and scoring code.

D.8 Evaluation Prompts

No LLM-as-a-judge prompt is used. Program validation, feasibility checking, objective evaluation, loss normalization, empirical mean computation, empirical CVaR_q computation, maximum-loss computation, and failure-rate computation are deterministic.

D.9 Ablation Prompt Variants

Purpose. The "no tail records in prompt" ablation removes high-loss diagnostic cases from the mutation prompt while keeping all other prompts, validators, budgets, populations, and held-out tests fixed.

Inputs. `{TASK_SPECIFICATION}`, `{FUNCTION_SIGNATURE}`, `{VALIDATION_CONSTRAINTS}`, `{PARENT_PROGRAM}`, `{PARENT_SCORE}`, `{MEAN_LOSS}`, `{CVAR_LOSS}`, `{MAX_LOSS}`, `{FAILURE_RATE}`, and `{TAIL_LEVEL}`.

Output. A single complete Python function implementing `{FUNCTION_SIGNATURE}`.

Purpose. Tail-level ablations with $q = 0.80$ and $q = 0.95$ use the main mutation prompt without changing the prompt structure. Only the value substituted into `{TAIL_LEVEL}` and the deterministic CVaR score used for ranking are changed.

Inputs. The same inputs as Figure 4, with `{TAIL_LEVEL}` set to either `0.80` or `0.95`.

Output. A single complete Python function implementing `{FUNCTION_SIGNATURE}`.

Purpose. The "no accumulated losses" ablation keeps the same mutation prompt as CVaR-RMD, but the statistics inserted into the prompt are computed from the current evaluation batch rather than from the accumulated loss multiset.

Inputs. The same inputs as Figure 4, with performance summaries computed from the non-accumulated batch losses.

Output. A single complete Python function implementing `{FUNCTION_SIGNATURE}`.

```
You are an expert Python programmer designing heuristic algorithms for combinatorial
optimization.

Task specification:
{TASK_SPECIFICATION}

Required function signature:
{FUNCTION_SIGNATURE}

Validation and execution constraints:
{VALIDATION_CONSTRAINTS}

Reference heuristic context, if useful:
{REFERENCE_HEURISTICS}

Write a complete Python heuristic for the task. The function must preserve the required
signature exactly, return a feasible solution in the required format, and run within the
execution limit. Prefer simple and robust logic over brittle special cases.

Return only the complete Python function. Do not include Markdown formatting, explanations,
tests, or example usage.
```

Figure 5: Direct LLM sampling prompt.

D.10 Output Formatting

All LLM code-generation prompts enforce the same output contract:

- The response must contain exactly one complete Python implementation of `{FUNCTION_SIGNATURE}`.
- The required function signature must be preserved exactly.
- The response must not contain Markdown code fences, prose explanations, test cases, example calls, or text outside the function definition.
- Helper logic must be placed inside the required function unless the harness explicitly permits additional top-level definitions.
- The function must return a solution object in the task-specific format described in `{TASK_SPECIFICATION}`.
- The function must avoid unsupported packages, file or network access, nondeterministic global state, excessive runtime, infeasible outputs, and interface violations.

Malformed outputs are retried up to three times using the same prompt. If parsing, compilation, interface validation, execution, or feasibility checking fails after retries, the candidate receives the fixed failure penalty ℓ_{fail} under the deterministic evaluation protocol.

E Case Study

We provide qualitative examples to illustrate how CVaR-RMD changes the type of heuristics selected by the outer loop. The goal of this case study is not to compare aggregate scores, but to inspect the generated programs and the failure modes that the selection objective encourages or suppresses. In all examples, the LLM is used only as a mutation operator: it proposes executable variants of a fixed heuristic interface, while the evaluator accepts or rejects candidates according to their behavior over heterogeneous instances, stochastic seeds, and problem scales.

Bin packing: avoiding brittle "perfect-fit" behavior. A representative mean-selected bin-packing heuristic produced a compact rule that prioritized immediate utilization. Its core decision was to place each item into the bin that would leave the smallest remaining capacity after insertion:

> *"For each item, scan all open bins and choose the feasible bin with minimum residual capacity. If no bin fits, open a new bin."*

The code was syntactically clean and often behaved well on regular instances where item sizes admitted many near-perfect combinations. However, inspection of its outputs revealed a brittle pattern. On instances containing a mixture of medium-sized and small items, the heuristic repeatedly consumed small residual spaces too early. Later medium items then failed to fit into existing bins, forcing the program to open additional bins. The failure was

```
You are an expert Python programmer designing robust heuristic algorithms for combinatorial
optimization.

Task specification:
{TASK_SPECIFICATION}

Required function signature:
{FUNCTION_SIGNATURE}

Validation and execution constraints:
{VALIDATION_CONSTRAINTS}

The following parent program is a working heuristic candidate:
{PARENT_PROGRAM}

Parent performance summary:
fitness score: {PARENT_SCORE}
mean loss: {MEAN_LOSS}
CVaR@{TAIL_LEVEL}: {CVAR_LOSS}
maximum loss: {MAX_LOSS}
failure rate: {FAILURE_RATE}

High-loss environments observed for this parent:
{TAIL_RECORDS}

Write a mutated version of the parent program. Preserve the required function signature
exactly. The new program should remain simple, deterministic given its inputs and seed, and
robust on the high-loss cases above. Avoid changes that risk infeasible outputs, exceptions,
excessive runtime, or interface violations.

Return only the complete Python function. Do not include Markdown formatting, comments
outside the function, explanations, tests, or example usage.

Parent performance summary:
fitness score: {PARENT_SCORE}    where {PARENT_SCORE} is empirical mean loss
mean loss: {MEAN_LOSS}
CVaR@{TAIL_LEVEL}: {CVAR_LOSS}
maximum loss: {MAX_LOSS}
failure rate: {FAILURE_RATE}
```

Figure 6: Mean-RMD mutation prompt difference. All other fields are identical to Figure 4.

not caused by invalid solutions; rather, the heuristic made locally attractive placements that created unfavorable residual-capacity profiles.

A CVaR-selected mutation preserved the same general structure but introduced an explicit slack-aware decision rule:

> *"Prefer the tightest feasible bin only when the resulting residual capacity is either very small or remains large enough to accommodate future medium items. Otherwise, choose a slightly looser bin to preserve reusable slack. Break ties by favoring bins whose residual capacities are closer to the current median residual capacity."*

The corresponding generated code maintained a small summary of open-bin residual capacities and computed a penalty for placements that created isolated unusable gaps. Qualitatively, this candidate no longer treated every tight fit as equally desirable. When faced with an item that could fit into several bins, it sometimes avoided the tightest placement if doing so would create a residual capacity unlikely to be useful later. This behavior is consistent with CVaR-RMD's selection pressure: candidates that occasionally collapse on adversarial item-size mixtures are less likely to survive, even if their average behavior is strong.

Capacitated vehicle routing: balancing route compactness and residual capacity. For CVRP, a typical mean-selected candidate resembled a nearest-feasible insertion heuristic. Given a partial solution, it selected the next customer primarily by geographic proximity:

> *"For each route, repeatedly append the nearest unrouted customer that fits the remaining vehicle capacity. When no feasible customer remains, close the route and start a new one from the depot."*

The resulting routes were easy to interpret: vehi-

```
You are an expert Python programmer improving heuristic algorithms for combinatorial
optimization.

Task specification:
{TASK_SPECIFICATION}

Required function signature:
{FUNCTION_SIGNATURE}

Validation and execution constraints:
{VALIDATION_CONSTRAINTS}

Previously discovered high-performing programs and their mean-performance summaries:
{ARCHIVE_PROGRAMS}

Write a new complete Python heuristic that preserves the required signature exactly. Reuse
useful algorithmic ideas from the archived programs, but do not copy a program verbatim if
a simpler or more robust variant is possible. The program must return feasible outputs and
avoid exceptions, timeouts, and interface violations.

Return only the complete Python function. Do not include Markdown formatting, explanations,
tests, or example usage.
```

Figure 7: Archive-conditioned prompt used by mean-selection evolutionary baselines.

cles greedily swept through local neighborhoods. Nevertheless, qualitative inspection showed a recurring tail failure. In clustered instances with uneven demand, the heuristic committed early to visually compact routes but left several high-demand customers until late in construction. These remaining customers were geographically close to existing routes but could no longer be inserted because the routes had insufficient residual capacity. The final solution then contained routes that made long depot returns for isolated customers.

A CVaR-selected variant modified the insertion priority:

> *"Score each feasible customer by combining distance, demand pressure, and the remaining capacity after insertion. Customers with high demand are promoted when delaying them would make later insertion difficult. If several candidates have similar distance, prefer the one that reduces future capacity fragmentation."*

The generated program did not abandon locality; nearby customers were still preferred. The important difference was that distance was no longer the only organizing principle. The candidate explicitly represented residual capacity as a scarce resource and penalized decisions that would strand large-demand customers. In the produced routes, high-demand customers were usually assigned before the route became nearly full, while smaller customers were used later as flexible fillers. This led to more stable behavior across instance families because the heuristic did not depend on the fortunate ordering of customer demands.

Flow-shop scheduling: replacing a single dispatching signal with bottleneck-aware priorities. In the flow-shop task, a mean-selected heuristic generated job sequences using a simple aggregate processing-time rule:

> *"Compute a priority score for each job from its total processing time. Schedule jobs with larger total processing time earlier, with a small adjustment for the first machine."*

This heuristic was transparent and often produced plausible schedules. However, its behavior was unstable when processing times were unbalanced across machines. For example, when a subset of jobs had moderate total processing time but was unusually heavy on a downstream machine, the heuristic tended to schedule them too late. The resulting schedule accumulated idle time before the bottleneck machine and then produced a long congested tail.

A CVaR-selected mutation introduced a more structured priority:

> *"For each job, estimate its contribution to early-machine release and downstream bottleneck load. Promote jobs that prevent bottleneck starvation, but delay jobs whose early-machine load would block many short jobs without relieving the bot-*

```
You are an expert Python programmer designing robust heuristic algorithms for combinatorial
optimization.

Task specification:
{TASK_SPECIFICATION}

Required function signature:
{FUNCTION_SIGNATURE}

Validation and execution constraints:
{VALIDATION_CONSTRAINTS}

The following parent program is a working heuristic candidate:
{PARENT_PROGRAM}

Parent performance summary:
fitness score: {PARENT_SCORE}
mean loss: {MEAN_LOSS}
CVaR@{TAIL_LEVEL}: {CVAR_LOSS}
maximum loss: {MAX_LOSS}
failure rate: {FAILURE_RATE}

Write a mutated version of the parent program. Preserve the required function signature
exactly. The new program should remain simple, deterministic given its inputs and seed, and
robust across diverse instances. Avoid changes that risk infeasible outputs, exceptions,
excessive runtime, or interface violations.

Return only the complete Python function. Do not include Markdown formatting, comments
outside the function, explanations, tests, or example usage.
```

Figure 8: Mutation prompt for the no-tail-records ablation.

tleneck. Recompute priorities after partial sequencing."

The generated code used normalized machine-wise processing times. It also updated job priorities during construction, so the dispatching rule could react to the partial sequence already built. Qualitatively, the schedules produced by this candidate were less dominated by a single global ordering. Jobs with downstream-heavy profiles were moved earlier when they helped keep the bottleneck machine active, while jobs that merely increased early congestion were postponed. This illustrates how tail-risk selection can favor heuristics that encode safeguards against specific pathological schedules.

Observed mutation pattern. Across tasks, we observed a common qualitative pattern in the programs selected by CVaR-RMD. Mean-based selection often retained concise greedy rules whose outputs looked strong on regular instances but depended heavily on favorable input structure. CVaR-RMD preserved the interpretable greedy backbone while adding small protective mechanisms: slack preservation in bin packing, demand-aware capacity management in CVRP, and bottleneck-aware dispatching in flow-shop scheduling. These changes were not arbitrary complexity. They were localized modifications that directly addressed rare but consequential failure modes exposed by evaluating each candidate over varied environments.

This case study supports the central claim of the paper: changing the outer-loop selection pressure can change the qualitative character of LLM-generated heuristics. CVaR-RMD does not require the language model to understand robustness explicitly. The evaluator repeatedly filters generated code through tail-risk behavior, causing robust design motifs to accumulate over evolutionary search.

```
You are an expert Python programmer designing robust heuristic algorithms for combinatorial
optimization.

Task specification:
{TASK_SPECIFICATION}

Required function signature:
{FUNCTION_SIGNATURE}

Validation and execution constraints:
{VALIDATION_CONSTRAINTS}

The following parent program is a working heuristic candidate:
{PARENT_PROGRAM}

Parent performance summary:
fitness score: {PARENT_SCORE}
mean loss: {MEAN_LOSS}
CVaR@{TAIL_LEVEL}: {CVAR_LOSS}
maximum loss: {MAX_LOSS}
failure rate: {FAILURE_RATE}

High-loss environments observed for this parent:
{TAIL_RECORDS}

Write a mutated version of the parent program. Preserve the required function signature
exactly. The new program should remain simple, deterministic given its inputs and seed, and
robust on the high-loss cases above. Avoid changes that risk infeasible outputs, exceptions,
excessive runtime, or interface violations.

Return only the complete Python function. Do not include Markdown formatting, comments
outside the function, explanations, tests, or example usage.

For the q = 0.80 ablation:
CVaR@{TAIL_LEVEL} = CVaR@0.80

For the q = 0.95 ablation:
CVaR@{TAIL_LEVEL} = CVaR@0.95
```

Figure 9: Prompt substitution for tail-level ablations.

```
You are an expert Python programmer designing robust heuristic algorithms for combinatorial
optimization.

Task specification:
{TASK_SPECIFICATION}

Required function signature:
{FUNCTION_SIGNATURE}

Validation and execution constraints:
{VALIDATION_CONSTRAINTS}

The following parent program is a working heuristic candidate:
{PARENT_PROGRAM}

Parent performance summary:
fitness score: {PARENT_SCORE}
mean loss: {MEAN_LOSS}
CVaR@{TAIL_LEVEL}: {CVAR_LOSS}
maximum loss: {MAX_LOSS}
failure rate: {FAILURE_RATE}

High-loss environments observed for this parent:
{TAIL_RECORDS}

Write a mutated version of the parent program. Preserve the required function signature
exactly. The new program should remain simple, deterministic given its inputs and seed, and
robust on the high-loss cases above. Avoid changes that risk infeasible outputs, exceptions,
excessive runtime, or interface violations.

Return only the complete Python function. Do not include Markdown formatting, comments
outside the function, explanations, tests, or example usage.
```

Figure 10: Prompt substitution for the no-accumulated-losses ablation.