CAP-EVOLVE: CONSTRAINT-AWARE ADAPTIVE PENALTIES FOR LLM-GUIDED EVOLUTIONARY SEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model-guided evolutionary search (LLM-ES) has shown promise for scientific and engineering optimization, yet existing approaches to constraint handling are either coarse or rely on task-specific designs, leaving general-purpose constraint handling for LLM-ES underdeveloped. We propose Constraint-aware Adaptive Penalty (CAP) to address this gap: it augments scalar program fitness with explicit constraint handling while preserving the underlying evolutionary search. Concretely, CAP retains informative infeasible programs and adapts penalties from signed violations and the evolving feasible frontier. Under idealized oracle assumptions, we establish convergence of CAP to a constrained optimum. For finite-budget search, we develop CAP-Evolve, which augments CAP with violation normalization and alpha-relaxed frontier updates and operates with an LLM-ES procedure as its inner search. Using AdaEvolve as this component, we evaluate CAP-Evolve on four explicitly constrained program-optimization tasks against representative constraint-handling baselines. CAP-Evolve attains the best mean result in six of seven task configurations, and its multi-constraint variants outperform all corresponding baselines.

1 INTRODUCTION

Large language model-guided evolutionary search (LLM-ES) uses an LLM to propose candidate programs from evaluated search history and improves them through an evolutionary loop. By searching directly over executable programs, LLM-ES provides a flexible approach to scientific and engineering optimization (Novikov et al., 2025; Khrulkov et al., 2025; Cemri et al., 2026). Existing LLM-ES systems have primarily targeted single-objective optimization, although general-purpose methods for multi-objective LLM-ES also exist (Wang et al., 2024; Hieu et al., 2026). By contrast, general-purpose explicit constraint handling for LLM-ES remains underdeveloped. This gap matters whenever candidate programs must satisfy explicit feasibility requirements in addition to achieving a strong objective value.

Current constrained LLM-ES practice most commonly uses hard rejection: any candidate violating a constraint is declared invalid and removed from normal competition. This rule is simple, but it discards potentially useful near-feasible programs and provides little guidance when feasible candidates are scarce (Novikov et al., 2025; Cemri et al., 2026).

Penalty-based constraint handling offers an alternative by retaining graded violation information and allowing infeasible candidates to remain useful to search. Figure 1 illustrates this motivation in one dimension. The feasible candidates found so far lie near a suboptimal local maximum, while promising infeasible candidates lie in a region where the raw objective is higher, adjacent to a better feasible solution. When only feasible candidates are retained, local evolutionary search may struggle to escape the current locally optimal region and reach that better solution. Meanwhile, optimizing the raw objective around the infeasible candidates moves them farther from feasibility. Penalization reshapes their scalar fitness, allowing nearby candidate generation to move toward the feasibility boundary and discover the better feasible solution.

A small body of LLM-ES work has adopted this penalty-based approach, but typically relies on fixed, task-specific coefficients chosen from human prior knowledge (Fazliani et al., 2026). These

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

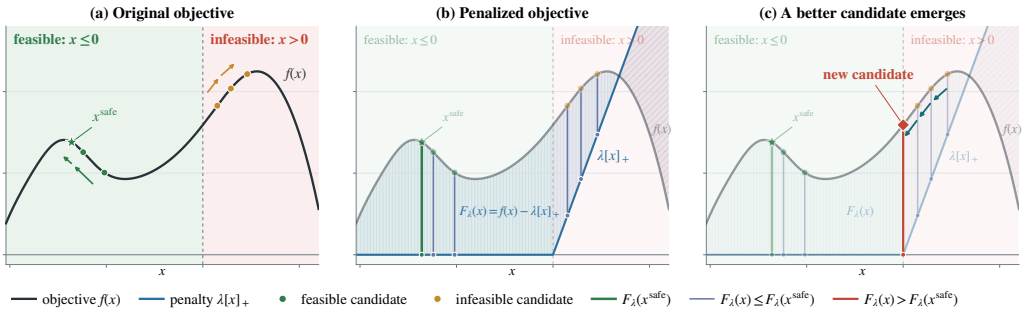


Figure 1: Motivation for penalty-based constraint handling.

coefficients depend on the objective scale and constraint design; in multi-constraint problems, hand-specifying a penalty vector also creates a combinatorial tuning problem. Thus, neither hard rejection nor existing fixed-penalty practice provides a general mechanism that adapts constraint handling to the evolving search archive.

To address this gap, we propose Constraint-aware Adaptive Penalty (CAP), a general constraint-handling mechanism for LLM-ES. Formally, CAP uses the best feasible candidate found so far as a frontier, treats infeasible candidates whose raw objective exceeds the frontier value as counterexamples, and updates a penalty vector so that their penalized fitness falls below that frontier. This update converts objective–violation feedback into a constraint-aware scalar fitness without prescribing a task-specific penalty vector. Under oracle assumptions, we prove convergence properties for CAP. For finite-budget LLM-ES, we implement CAP with a finite-round AdaEvolve (Cemri et al., 2026) search and add practical heuristics, yielding CAP-Evolve while retaining AdaEvolve's underlying search framework.

We evaluate CAP-Evolve on a unified suite of explicitly constrained program-optimization tasks spanning mathematical and engineering domains (Blind & Ament, 2000; Bucur, 2012; Rastgar et al., 2021; Paruchuri et al., 2025). The suite covers both open-loop solution construction and closed-loop policy construction, as well as single- and multi-constraint settings. CAP-Evolve performs best in six of seven task configurations; in particular, its multi-constraint variants outperform all corresponding baselines.

Together, these components make the following contributions:

1. We propose CAP, a general constraint-handling mechanism for LLM-ES, and establish finite-time discovery of a constrained optimum and batch-oracle asymptotic optimality for its idealized frontier-based update under explicit assumptions.
2. We develop CAP-Evolve, a practical finite-budget instantiation of CAP for executable program search, implemented within AdaEvolve with active-count normalization and alpha-relaxed frontier updates.
3. We provide a unified suite for explicitly constrained program optimization and evaluate CAP-Evolve against hard-rejection, fixed-penalty, and adaptive-penalty baselines under a common protocol, with CAP-Evolve performing best in six of seven task configurations and its multi-constraint variants outperforming all corresponding baselines.

2 Related Work

LLM-ES, represented by systems such as AlphaEvolve (Novikov et al., 2025), combines language-model generation with evolutionary search over executable artifacts. A typical iteration selects parents and context, generates candidate mutations, evaluates them with a task-specific evaluator, and updates the search state. Recent work improves each stage of this workflow. ShinkaEvolve (Lange et al., 2025) studies parent sampling and novelty-based rejection for candidate selection. Candidate generation is improved through adaptive model selection in AdaptEvolve (Ray et al., 2026), test-time model adaptation in ThetaEvolve (Wang et al., 2025), and directed generation in LoongFlow (Wan

et al., 2025). GigaEvo (Khrulkov et al., 2025) develops asynchronous evaluation for large-scale search. CodeEvolve (Assumpção et al., 2026) and AdaEvolve (Cemri et al., 2026) improve population and archive management through quality-diversity and adaptive exploration mechanisms, respectively. Other work optimizes the prompts that guide candidate generation through evolutionary and reflective prompt search (Guo et al., 2024; Agrawal et al., 2026).

Constrained optimization is common across scientific and engineering applications (Zimmerman et al., 2011; Markowitz, 1952; Kumar et al., 2020). It also arises in LLM-ES benchmarks such as circle packing (Blind & Ament, 2000; Novikov et al., 2025; Cemri et al., 2026) and aerodynamic shape optimization (Fazliani et al., 2026). Existing LLM-ES systems most commonly handle such constraints through evaluator-level rejection (Novikov et al., 2025; Cemri et al., 2026), while some task-specific systems use fixed terminal scores (Wang et al., 2025) or fixed penalties (Fazliani et al., 2026). These approaches either discard graded violation information or rely on fixed penalty weights that do not adapt to the evolving search archive.

More broadly, optimization research has studied constraint handling through feasibility filtering, penalty or Lagrangian relaxation, and repair or projection (Coello Coello, 2002; Mezura-Montes & Coello Coello, 2011). Filtering can use constraint signals only to produce a feasibility ordering (Kramer, 2010), which does not match the scalar-score interface of most LLM-ES systems. Repair or projection requires a map from infeasible points to the feasible set (Salcedo-Sanz, 2009); constructing an inexpensive, semantics-preserving map is extremely difficult in program space. Penalty-based methods instead aggregate the objective and constraint violations into a scalar fitness and can therefore be integrated into existing single-objective search procedures with limited modification. CAP follows this route while inferring its penalty vector adaptively from the evolving search archive rather than relying on hand-specified fixed weights.

3 CAP: Formulation and Theory

3.1 Problem Formulation

Let $\mathcal{X}$ be the search space. Evaluating $x \in \mathcal{X}$ returns a raw objective $f(x)$ and a K-dimensional signed violation vector $\mathbf{v}(x) = (v_1(x), \ldots, v_K(x))$. Constraint i is satisfied if and only if $v_i(x) \leq 0$; overall feasibility means $\mathbf{v}(x) \leq 0$ componentwise, defining the feasible set $\mathcal{F} = \{x \in \mathcal{X} : \mathbf{v}(x) \leq 0\}$. We express all tasks as maximization without loss of generality. The general form of the constrained program-optimization problem considered in this work is

$$\max_{x \in \mathcal{X}} f(x) \qquad \text{s.t.} \qquad v_i(x) \leq 0, \quad i = 1, \ldots, K.$$

3.2 CAP Update

CAP is an adaptive-penalty method that converts a single-objective constrained optimization problem into a sequence of single-objective unconstrained search problems. At each update, CAP constructs a scalar penalized objective from the raw objective and constraint violations. Unlike a fixed penalty, its penalty vector is inferred from the current search archive through a quadratic program (QP), so that the scalar objective changes as the observed feasible frontier and infeasible counterexamples evolve.

Given a nonnegative penalty vector $\boldsymbol{\lambda} \in \mathbb{R}^K_+$, CAP maps each candidate to the scalar fitness

$$F_{\boldsymbol{\lambda}}(x) = f(x) - \boldsymbol{\lambda}^\top \mathbf{u}(x),$$

where $\mathbf{u}(x) = [\mathbf{v}(x)]_+$ is the positive violation vector. Feasible programs have zero penalty, so their scalar fitness equals their raw objective. For an archive X with $X \cap \mathcal{F} \neq \varnothing$, the safe frontier is the feasible candidate $x^{\text{safe}} = \arg\max_{x \in X \cap \mathcal{F}} f(x)$. Historical infeasible candidates whose raw objective exceeds the frontier value provide counterexamples that the penalty should suppress.

For each such counterexample y, CAP requires its penalized score to lie below the safe frontier, namely $F_{\boldsymbol{\lambda}}(y) \leq F_{\boldsymbol{\lambda}}(x^{\text{safe}})$. Because x^{safe} is feasible, this is equivalent to requiring

$$f(x^{\text{safe}}) \geq f(y) - \boldsymbol{\lambda}^\top \mathbf{u}(y).$$

CAP updates $\boldsymbol{\lambda}$ to a minimum-norm nonnegative vector satisfying the accumulated frontier inequalities. Thus, the update chooses the weakest penalty that assigns every observed counterexample a scalar fitness no greater than that of the feasible frontier. Figure 2 summarizes the resulting loop. Evaluated candidates populate the archive X_i with their objective and violation feedback; the archive then supplies the safe frontier and counterexamples used by the Frontier QP to obtain $\boldsymbol{\lambda}_i$. Combining this penalty with the original problem $(f, \mathbf{v})$ constructs $F_{\boldsymbol{\lambda}_i}$, which, together with X_i, drives parent selection, prompt building, and candidate generation in the LLM-ES inner search. The resulting batch B_i is evaluated on the constrained problem and fed back into the archive for the next cycle.

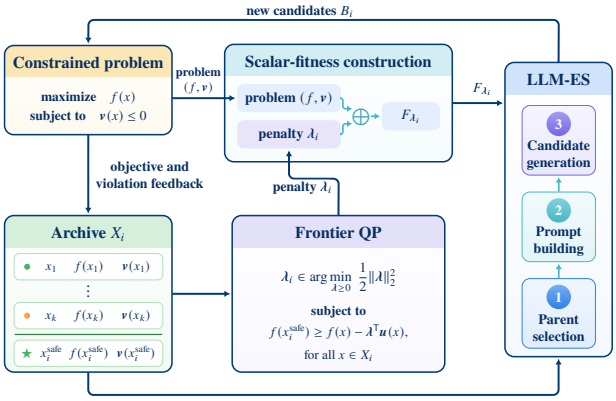


Figure 2: CAP integration with LLM-ES.

Definition 1 (CAP)**.** For any finite archive $X \subset \mathcal{X}$ with $X \cap \mathcal{F} \neq \varnothing$, define

$$x^{\text{safe}}(X) \in \arg\max_{x \in X \cap \mathcal{F}} f(x)$$

and the archive-to-penalty map

$$\begin{aligned} \mathcal{P}(X) := \arg\min_{\boldsymbol{\lambda} \in \mathbb{R}_+^K} \quad & \tfrac{1}{2}\|\boldsymbol{\lambda}\|_2^2 \\ \text{s.t.} \quad & f\big(x^{\text{safe}}(X)\big) \geq f(x) - \boldsymbol{\lambda}^\top \mathbf{u}(x), \qquad \forall x \in X. \end{aligned} \tag{1}$$

Given a finite initial archive X_0 with $X_0 \cap \mathcal{F} \neq \varnothing$ and an inner-search operator $\mathcal{A}$ returning a nonempty finite subset of $\mathcal{X}$, CAP is defined by the infinite recursion

$$B_i := \mathcal{A}(F_{\boldsymbol{\lambda}_i}), \qquad X_{i+1} := X_i \cup B_i, \qquad \boldsymbol{\lambda}_{i+1} := \mathcal{P}(X_{i+1}), \tag{2}$$

initialized by $\boldsymbol{\lambda}_0 := \mathcal{P}(X_0)$, with $x_i^{\text{safe}} := x^{\text{safe}}(X_i)$.

The resulting CAP update is well suited to finite-budget LLM-ES: it provides archive-conditioned candidate-level feedback, controls its numerical scale through minimum-norm penalty selection, and responds directly as new programs enter the archive. Appendix A discusses how these properties support the short, feedback-driven search process of LLM-ES.

3.3 Theory Overview

The analysis specializes Definition 1 to an idealized continuous search space $\mathcal{X} \subset \mathbb{R}^n$. Let f^* denote the optimal constrained objective and define the global exact-penalty set $\Lambda^* = \{\boldsymbol{\lambda} \in \mathbb{R}_+^K : F_{\boldsymbol{\lambda}}(x) \leq f^*, \ \forall x \in \mathcal{X}\}$.

Assumption 1. *The search space $\mathcal{X} \subset \mathbb{R}^n$ is nonempty and compact; $f : \mathcal{X} \to \mathbb{R}$ and $\mathbf{v} : \mathcal{X} \to \mathbb{R}^K$ are continuous; and the feasible set $\mathcal{F} = \{x \in \mathcal{X} : \mathbf{v}(x) \leq 0\}$ is nonempty. The initial archive $X_0 \subset \mathcal{X}$ is finite and contains a feasible point. The global exact-penalty set Λ^* is nonempty.*

The final condition in Assumption 1, $\Lambda^* \neq \varnothing$, is nontrivial but can be verified through sufficient conditions. Theorem B.1 gives one such condition based on Lipschitz continuity of the objective and a global error bound for the constraint system.

Definition 2 (Asymptotic direction coverage)**.** A nonnegative penalty sequence $\{\boldsymbol{\lambda}_i\}$ has asymptotic direction coverage if

$$\limsup_{i\to\infty} \|\boldsymbol{\lambda}_i\|_2 = \infty \quad \Longrightarrow \quad \limsup_{i\to\infty} \min_{j=1,\dots,K} (\boldsymbol{\lambda}_i)_j = \infty.$$

The following results characterize oracle CAP. The first establishes that a constrained global optimum appears in the archive after finitely many iterations when each iteration returns a single penalized global maximizer; the second establishes asymptotic convergence of the best feasible archive value when each iteration returns a finite batch containing such a maximizer.

Theorem 1 (Finite-time discovery by oracle CAP)**.** *Under Assumption 1, consider CAP in Definition 1 with $B_i = \{y_i\}$, where $y_i \in \arg\max_{x\in\mathcal{X}} F_{\boldsymbol{\lambda}_i}(x)$ at every iteration. If its penalty sequence has asymptotic direction coverage, then X_i contains a globally optimal solution of the constrained problem for some finite i.*

Theorem 2 (Asymptotic optimality of batch-oracle CAP)**.** *Under Assumption 1, consider CAP in Definition 1 with $B_i \cap \arg\max_{x\in\mathcal{X}} F_{\boldsymbol{\lambda}_i}(x) \neq \varnothing$ at every iteration. If its penalty sequence has asymptotic direction coverage, then its best feasible archive value converges to the constrained optimum.*

Proofs are given in Appendix B.

4 CAP-EVOLVE

LLM-ES integration. CAP-Evolve applies CAP to executable program search by implementing the inner-search operator $\mathcal{A}$ in Definition 1 with a finite-budget LLM-ES component. The LLM-ES generates and evaluates programs under the current scalar fitness, while CAP periodically updates that fitness from their objectives and signed violations. Our implementation uses AdaEvolve (Cemri et al., 2026), an island-structured LLM-ES method that maintains multiple interacting local populations; Appendix C summarizes the underlying workflow and this island organization. Implementing this integration under a finite budget requires practical choices about violation normalization, penalty scope, archive maintenance, and update scheduling. We describe these choices below; Appendix D.1 provides a direct comparison with idealized CAP.

Violation normalization. CAP-Evolve first maps signed violations to a practical scalar fitness. Let $a(x) = \|[\mathbf{v}(x)]_+\|_0$ be the number of active constraints. For an exponent $\beta \geq 0$, define the active-count normalization

$$\tilde{\mathbf{u}}_\beta(x) = \frac{[\mathbf{v}(x)]_+}{\max\{1, a(x)^\beta\}}.$$

The choices $\beta = 0$, $\beta = 1$, and $\beta = \frac{1}{2}$ give no normalization, active-count mean normalization, and active-count square-root normalization, respectively. CAP-Evolve uses $\beta = \frac{1}{2}$ in the main experiments, with

$$\tilde{\mathbf{u}}(x) = \tilde{\mathbf{u}}_{1/2}(x), \qquad \widetilde{F}_{\boldsymbol{\lambda}}(x) = f(x) - \boldsymbol{\lambda}^\top \tilde{\mathbf{u}}(x).$$

The same normalized signal is used in both fitness evaluation and penalty updates. It preserves feasibility while reducing sensitivity to whether a condition is represented by one aggregate constraint or several simultaneously active constraints. In particular, the ℓ_2 scale of a row with similarly sized active violations is approximately invariant to the number of such violations. The square-root factor retains a sublinear penalty increase when more constraints are simultaneously violated.

Island-specific penalty scope. For each AdaEvolve island s, CAP-Evolve maintains an island-specific penalty vector $\boldsymbol{\lambda}_s$ and constraint-aware archive state. This allows islands to follow distinct objective–constraint trade-offs during search. Penalty updates are triggered every $h = 5$ AdaEvolve iterations. Whenever CAP updates $\boldsymbol{\lambda}_s$, it recomputes the constraint-aware fitness of all retained

candidates in island s before AdaEvolve resumes selection. This ensures that each island archive is evaluated under one consistent scalarization.

Alpha relaxation. For an island with a feasible frontier, CAP-Evolve applies alpha relaxation to the frontier QP in Equation 1. Let N be the number of eligible counterexample rows. At AdaEvolve iteration t, CAP-Evolve ranks each row y by the difficulty score $d_y = (f(y) - f(x^{\text{safe}}))/\|\tilde{\mathbf{u}}(y)\|_2$, and lets R_{α_t} denote the rows remaining after removing the $\lfloor \alpha_t N \rfloor$ rows with the largest scores, while retaining at least one row when $N > 0$. CAP computes

$$\begin{aligned} \boldsymbol{\lambda} \in \arg\min_{\boldsymbol{\lambda} \geq 0} \quad & \tfrac{1}{2}\|\boldsymbol{\lambda}\|_2^2 \\ \text{s.t.} \quad & f(x^{\text{safe}}) \geq f(y) - \boldsymbol{\lambda}^\top \tilde{\mathbf{u}}(y), \quad y \in R_{\alpha_t}. \end{aligned} \tag{3}$$

The relaxation fraction α_t decreases linearly from α_0 to zero by iteration 80, at which point every eligible row enters the QP. Early in the search, this allows selected infeasible candidates to remain competitive with the best feasible candidate, providing an exploration–exploitation trade-off that gradually shifts toward stricter constraint enforcement.

Overall procedure. Algorithm 1 summarizes the combined CAP-Evolve loop at the granularity of CAP updates. In each outer step, the stateful AdaEvolve (Cemri et al., 2026) routine `AdaEvolveGenerate` executes h sequential iterations with internal island updates and returns the evaluated candidate batch B_k, including island assignments; `AdaEvolveUpdateIslands` then synchronizes the retained island states. CAP invokes `AlphaRelax` for difficulty-based row removal, solves the resulting frontier QP, and rescores the affected archives before the next AdaEvolve block. Scheduling details, numerical safeguards, and parameter settings are provided in Appendix D.

Algorithm 1 CAP-Evolve: alternating blocks of AdaEvolve search and frontier-based penalty updates.

Require: Islands $\{X_{s,0}\}_{s=1}^{S}$ with $X_{s,0} \cap \mathcal{F} \neq \varnothing$; initial penalties $\{\boldsymbol{\lambda}_{s,0}\}$; evaluator E; AdaEvolve-iteration budget T; update interval h with $U = T/h$; schedule $\{\alpha_t\}$.

for all $s = 1, \ldots, S$ **do**
  $X_s \leftarrow X_{s,0}$; $\boldsymbol{\lambda}_s \leftarrow \boldsymbol{\lambda}_{s,0}$; assign $\widetilde{F}_{\boldsymbol{\lambda}_s}(x)$ to every $x \in X_s$.
end for
for $k = 1, \ldots, U$ **do**
  $t \leftarrow kh$; $B_k \leftarrow$ AdaEvolveGenerate$(\{X_s, \widetilde{F}_{\boldsymbol{\lambda}_s}\}_{s=1}^{S}, E, h)$.
  $\{X_s\}_{s=1}^{S} \leftarrow$ AdaEvolveUpdateIslands$(\{X_s\}_{s=1}^{S}, B_k)$.
  for all islands s **do**
    $x_s^{\text{safe}} \leftarrow \arg\max\{f(x) : x \in X_s, \mathbf{v}(x) \leq 0\}$.
    $R_s \leftarrow \{y \in X_s : \mathbf{v}(y) \not\leq 0, f(y) > f(x_s^{\text{safe}})\}$; $R_{s,\alpha_t} \leftarrow$ AlphaRelax(R_s, α_t).
    if $R_{s,\alpha_t} \neq \varnothing$ **then**
      Solve Equation 3 for $\boldsymbol{\lambda}_s$ using x_s^{safe} and R_{s,α_t}; rescore all $x \in X_s$.
    end if
  end for
end for
return $\arg\max\{f(x) : x \in \bigcup_s X_s, \mathbf{v}(x) \leq 0\}$.

5 Experimental Evaluation

5.1 Evaluation Suite

We evaluate CAP-Evolve on four complementary constrained program-optimization tasks. Circle Packing is a standard LLM-ES benchmark with explicit geometric constraints and allows us to study multiple equivalent constraint representations (Blind & Ament, 2000; Novikov et al., 2025). Spectral Shape Optimization considers scientific shape construction under a global area constraint (Bucur, 2012). Multi-Agent Trajectory Optimization tests open-loop program construction with endpoint

and collision constraints across multiple environments (Rastgar et al., 2021), whereas Tokamak Control tests closed-loop policy construction subject to a safety condition over simulated episodes (Paruchuri et al., 2025). For tasks represented with multiple constraint counts, the alternative representations expose different violation information but define the same feasible set. Together, the suite spans mathematical and engineering domains, static construction and dynamical control, and single- and multi-constraint settings. Figure 3 visualizes the four tasks; complete benchmark specifications are provided in Appendix F.

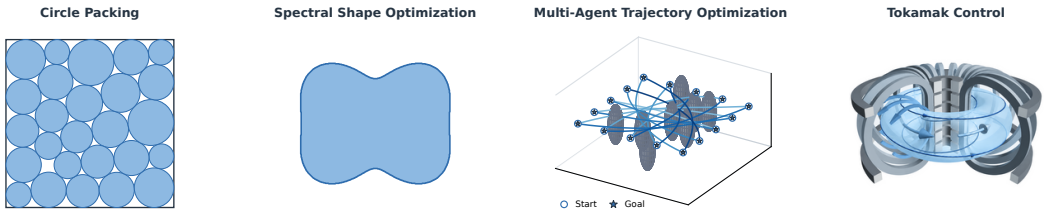


Figure 3: Overview of the four evaluation tasks.

5.2 Compared Methods and Protocol

Our main comparison includes four handling rules. AdaEvolve (Cemri et al., 2026) with hard rejection excludes infeasible candidates from normal competition; because equivalent representations induce the same feasibility test, we use one shared hard-rejection run across the reported representations. Fixed-penalty AdaEvolve is evaluated only on single-constraint tasks, using a common power-of-ten grid $\lambda = 10^n$ and reporting the coefficient with the best mean best-feasible task-native metric across five runs; complete sweeps and selection conventions are reported in Appendix H. The Tessema–Yen (T–Y) adaptive penalty (Tessema & Yen, 2006) constructs a population-dependent scalar fitness from the normalized objective and aggregate constraint violation without a user-specified penalty coefficient; its adaptation and implementation are detailed in Appendix E. CAP-Evolve uses the task-specific α_0 selected by the mean best-feasible criterion from $\{0.25, 0.5, 0.75\}$, with the selected values reported in Appendix D.

All compared methods use the same initial program, LLM, task evaluator, 100-iteration search budget per run, and five independent runs. Every candidate evaluation is subject to the same 660-second wall-clock limit, including candidate-internal simulation, numerical optimization, and evaluator calls. Thus, all methods use the same nominal number of evolutionary iterations and the same maximum execution budget per candidate; individual programs may finish before that limit. To control for the substantial effect of prompting, the methods share a common task prompt and AdaEvolve generation template; method-specific edits are restricted to the instructions required by the respective constraint-handling rule. At the end of each run, we select the retained feasible program with the best raw objective.

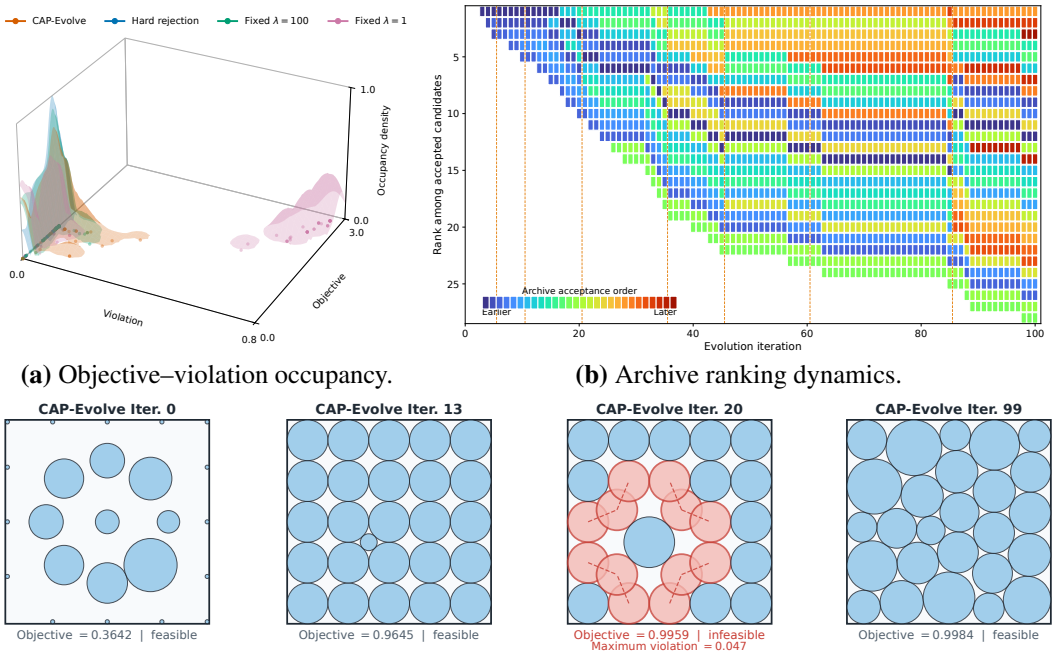
5.3 Main Comparison

Entries report the mean and standard deviation of the task-native, best-feasible metric over five independent runs, with complete run-level results in Appendix G. For readability, trajectory and tokamak results are reported with sign-reversed metrics, so lower values indicate better performance; these metrics are equivalent to the maximization objectives used during search. Arrows indicate the preferred direction. The T–Y adaptive penalty underperforms CAP-Evolve across all configurations and the selected fixed penalty wherever the latter is evaluated. This result shows that conventional evolutionary constraint handling does not transfer reliably to LLM-ES without accounting for its distinct search dynamics. CAP-Evolve performs best in six of seven task configurations. Its multi-constraint variants outperform all corresponding baselines, with the largest margins appearing in the fine-grained 26-constraint Circle Packing representation and the five-constraint trajectory setting. In the single-constraint settings, CAP-Evolve remains competitive with the selected fixed penalties.

Table 1: Main comparison under a 100-iteration budget.

Task	K	AdaEvolve (hard rejection)	AdaEvolve (best fixed penalty)	AdaEvolve (T–Y penalty)	CAP-Evolve
Circle Packing ↑	1	0.98606±0.02262	0.99598±0.00354	0.98463±0.01637	**0.99647±0.00308**
	3		–	0.99520±0.00423	**0.99561±0.00451**
	26		–	0.99009±0.01459	**0.99848±0.00090**
Spectral Shape Optimization ↑	1	1.0108±0.01120	1.0220±0.01920	1.0113±0.00450	**1.0231±0.02450**
Trajectory Optimization ↓	1	1.35797±0.00508	1.35924±0.00260	1.35951±0.00281	1.35934±0.00242
	5		–	1.35816±0.00387	**1.35494±0.00583**
Tokamak Control ↓	1	0.06656±0.00754	0.06526±0.00175	0.07647±0.02805	**0.06517±0.00324**

5.4 Case Study



(a) Objective–violation occupancy. **(b)** Archive ranking dynamics.

(c) Evolution states in a run.

Figure 4: Representative search behavior.

Figure 4 uses distinct representative Circle Packing runs to illustrate complementary aspects of CAP-Evolve. Panel (a) compares objective–violation occupancy across methods in the single-constraint setting. Hard rejection remains on the feasible boundary; the fixed penalties show strong coefficient dependence: $\lambda = 100$ concentrates near feasibility, whereas $\lambda = 1$ reaches higher raw objective values at the cost of large violations. CAP-Evolve covers the feasible boundary and a broader range of informative violations. Panel (b) follows candidates accepted into one island during a $K = 26$ CAP-Evolve run. Each column gives the candidates' scalar-fitness ranks at one iteration, and each orange dashed line marks a recomputation that changes the value of $\boldsymbol{\lambda}$. The rank changes around some of these updates illustrate how an updated penalty can rescore previously generated candidates as search proceeds.

Panel (c) traces representative states from a single-constraint CAP-Evolve run. The initial feasible packing is substantially improved by iteration 13. Its direct child at iteration 20 attains a higher raw objective but is infeasible and is subsequently included in multiple frontier computations. A

separate lineage branching from the iteration-13 solution eventually produces its best descendant at iteration 99, which is feasible and further improves the objective. This case illustrates how CAP-Evolve can retain informative infeasible candidates for frontier updates while continuing to improve feasible lineages.

5.5 Ablations

We perform component ablations under the three constraint representations of Circle Packing, varying the normalization exponent β, the initial alpha relaxation fraction, and the penalty scope. The normalization study compares $\beta \in \{0, \frac{1}{2}, 1\}$, corresponding to no, square-root, and mean normalization, respectively; the alpha study compares the relaxed and non-relaxed frontier updates; and the penalty-scope study compares island-specific penalties with a single globally shared penalty. The results are reported in Table 2.

Table 2: Circle Packing ablations under a 100-iteration budget.

K	Alpha relaxation	Violation normalization	Penalty scope	Target ratio ↑
1	With	Sqrt	Island	$\mathbf{0.99647 \pm 0.00308}$
	With	Sqrt	Global	0.99118 ± 0.00780
	Without	Sqrt	Island	0.99567 ± 0.00310
3	With	Sqrt	Island	0.99561 ± 0.00451
	With	Sqrt	Global	0.99485 ± 0.00370
	Without	Sqrt	Island	0.99353 ± 0.00321
	With	None	Island	$\mathbf{0.99684 \pm 0.00270}$
	With	Mean	Island	0.99486 ± 0.00500
26	With	Sqrt	Island	$\mathbf{0.99848 \pm 0.00090}$
	With	Sqrt	Global	0.99422 ± 0.00430
	Without	Sqrt	Island	0.98534 ± 0.02186
	With	None	Island	0.99187 ± 0.00678
	With	Mean	Island	0.99035 ± 0.01126

The benefit of square-root normalization is most evident for the higher-dimensional $K = 26$ representation, where it improves both the mean and variability. Its effect is smaller for the lower-dimensional representations and is absent when $K = 1$, where all normalization choices are equivalent. Alpha relaxation improves the mean result at all three representation granularities; its clearest effect is at $K = 26$, where removing it substantially lowers the mean target ratio and increases variation. Island-specific penalties improve the mean result across all three representations, although the gain is smaller at $K = 3$ than at $K = 1$ or $K = 26$. This suggests that maintaining separate penalties can be beneficial when different islands encounter distinct objective–constraint trade-offs. Complete run-level component results, alpha sensitivity studies, and fixed-penalty sweeps are reported in Appendix H.

6 Discussion and Conclusion

We studied general constraint handling for LLM-ES, a problem that existing systems typically address through coarse rejection rules or task-specific designs. We proposed CAP, which retains informative infeasible candidates and adapts a scalar penalty from signed violations and the evolving feasible frontier. We established convergence properties for CAP under oracle assumptions and instantiated the framework in finite-budget program search as CAP-Evolve, incorporating normalization and alpha-relaxed frontier updates. Across four constrained program-optimization tasks, CAP-Evolve performs best in six of seven configurations, and its multi-constraint variants outperform all corresponding baselines. These results provide an initial theoretical and empirical foundation for constrained optimization in LLM-ES. Future work can investigate alternative algorithmic instantiations within the CAP framework and develop new constraint-handling principles that explicitly account for approximate search, finite budgets, and evolving populations.

AI USE STATEMENT

In this work, we used generative AI tools to help formalize and refine the written proofs of mathematical claims developed by the authors. We also used generative AI tools to help implement experimental code and statistically summarize experimental results. We did not use generative AI to develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, propose or refine hypotheses, design or provide feedback on research methodology or experiments, assist with translation, support qualitative or thematic data analysis, or interpret results. Synthetic-dataset generation and dataset cleaning or reformatting are not applicable to this work. Additionally, we used generative AI to create and edit scientific figures and to polish writing. We reviewed all AI-assisted work and take responsibility for the final content of this work, including all claims, results, code, and artifacts produced with the aid of generative AI.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 8479–8565, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/0e9e708b6f48e14fd0ac29e167413f76-Paper-Conference.pdf.

Henrique Assumpção, Diego Ferreira, Leandro Campos, and Fabricio Murai. Codeevolve: an open source evolutionary coding agent for algorithmic discovery and optimization, 2026. URL https://arxiv.org/abs/2510.14150.

G. Blind and P. Ament. Packing equal circles in a square. *Studia Scientiarum Mathematicarum Hungarica*, 36(3-4):313316, December 2000. ISSN 1588-2896. doi: 10.1556/sscmath.36.2000.3-4.3. URL http://dx.doi.org/10.1556/SSCMATH.36.2000.3-4.3.

Dorin Bucur. Minimization of the k-th eigenvalue of the dirichlet laplacian. *Archive for Rational Mechanics and Analysis*, 206(3):1073–1083, September 2012. ISSN 1432-0673. doi: 10.1007/s00205-012-0561-0. URL http://dx.doi.org/10.1007/s00205-012-0561-0.

Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren, Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, Alex Dimakis, and Ion Stoica. Adaevolve: Adaptive llm driven zeroth-order optimization, 2026. URL https://arxiv.org/abs/2602.20133.

Carlos A Coello Coello. Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering*, 191(11-12):12451287, January 2002. ISSN 0045-7825. doi: 10.1016/s0045-7825(01)00323-1. URL http://dx.doi.org/10.1016/S0045-7825(01)00323-1.

Shaghayegh Fazliani, Krissh Chawla, Jack Guo, Yiren Shen, Matthias Ihme, and Madeleine Udell. Shapebench: A scalable benchmark and diagnostic suite for standardized evaluation in aerodynamic shape optimization, 2026. URL https://arxiv.org/abs/2605.20763.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 34133–34156, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/9156b0f6dfa9bbd18c79cc459ef5d61c-Paper-Conference.pdf.

Ha Minh Hieu, Hung Phan, Tung Duy Doan, Tung Dao, Cong Dao Tran, and Huynh Thi Thanh Binh. Pareto-grid-guided large language models for fast and high-quality heuristics design in multi-objective combinatorial optimization. *Proceedings of the AAAI Conference on Artificial*

Intelligence, 40(43):36964–36972, March 2026. ISSN 2159-5399. doi: 10.1609/aaai.v40i43.41024. URL http://dx.doi.org/10.1609/aaai.v40i43.41024.

Valentin Khrulkov, Andrey Galichin, Denis Bashkirov, Dmitry Vinichenko, Oleg Travkin, Roman Alferov, Andrey Kuznetsov, and Ivan Oseledets. Gigaevo: An open source optimization framework powered by llms and evolution algorithms, 2025. URL https://arxiv.org/abs/2511.17592.

Oliver Kramer. A review of constraint-handling techniques for evolution strategies. *Applied Computational Intelligence and Soft Computing*, 2010:111, 2010. ISSN 1687-9732. doi: 10.1155/2010/185063. URL http://dx.doi.org/10.1155/2010/185063.

Abhishek Kumar, Guohua Wu, Mostafa Z. Ali, Rammohan Mallipeddi, Ponnuthurai Nagaratnam Suganthan, and Swagatam Das. A test-suite of non-convex constrained optimization problems from the real-world and some baseline results. *Swarm and Evolutionary Computation*, 56:100693, August 2020. ISSN 2210-6502. doi: 10.1016/j.swevo.2020.100693. URL http://dx.doi.org/10.1016/j.swevo.2020.100693.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution, 2025. URL https://arxiv.org/abs/2509.19349.

Harry Markowitz. Portfolio selection. *The Journal of Finance*, 7(1):77, March 1952. ISSN 0022-1082. doi: 10.2307/2975974. URL http://dx.doi.org/10.2307/2975974.

Efrén Mezura-Montes and Carlos A. Coello Coello. Constraint-handling in nature-inspired numerical optimization: Past, present and future. *Swarm and Evolutionary Computation*, 1 (4):173–194, December 2011. ISSN 2210-6502. doi: 10.1016/j.swevo.2011.10.001. URL http://dx.doi.org/10.1016/j.swevo.2011.10.001.

Alexander Novikov, Ngân V, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

Sai Tej Paruchuri, Ian Ward, Nicholas Rist, Vincent Graber, Hassan Al Khawaldeh, Zibo Wang, Eugenio Schuster, Andres Pajares, and June-Woo Juhn. Density regulation with disruption avoidance in next-generation tokamaks using a safe reinforcement learning-based controller. *Fusion Engineering and Design*, 216:115064, July 2025. ISSN 0920-3796. doi: 10.1016/j.fusengdes.2025.115064. URL http://dx.doi.org/10.1016/j.fusengdes.2025.115064.

Fatemeh Rastgar, Houman Masnavi, Jatan Shrestha, Karl Kruusamae, Alvo Aabloo, and Arun Kumar Singh. Gpu accelerated convex approximations for fast multi-agent trajectory optimization. *IEEE Robotics and Automation Letters*, 6(2):3303–3310, April 2021. ISSN 2377-3774. doi: 10.1109/lra.2021.3061398. URL http://dx.doi.org/10.1109/LRA.2021.3061398.

Pretam Ray, Pratik Prabhanjan Brahma, Zicheng Liu, and Emad Barsoum. Adaptevolve: Improving efficiency of evolutionary ai agents through adaptive model selection. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 40625–40633. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.2019. URL https://doi.org/10.18653/v1/2026.findings-acl.2019.

Sancho Salcedo-Sanz. A survey of repair methods used as constraint handling techniques in evolutionary algorithms. *Computer Science Review*, 3(3):175–192, August 2009. ISSN 1574-0137. doi: 10.1016/j.cosrev.2009.07.001. URL http://dx.doi.org/10.1016/j.cosrev.2009.07.001.

Biruk G. Tessema and G. Yen. A self adaptive penalty function based algorithm for constrained optimization. *2006 IEEE International Conference on Evolutionary Computation*, pp. 246–253, 2006. doi: 10.1109/CEC.2006.1688315.

Chunhui Wan, Xunan Dai, Zhuo Wang, Minglei Li, Yanpeng Wang, Yinan Mao, Yu Lan, and Zhiwen Xiao. Loongflow: Directed evolutionary search via a cognitive plan-execute-summarize paradigm, 2025. URL https://arxiv.org/abs/2512.24077.

Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, Hao Cheng, Pengcheng He, Weizhu Chen, Shuohang Wang, Simon Shaolei Du, and Yelong Shen. Thetaevolve: Test-time learning on open problems, 2025. URL https://arxiv.org/abs/2511.23473.

Zeyi Wang, Songbai Liu, Jianyong Chen, and Kay Chen Tan. *Large Language Model-Aided Evolutionary Search for Constrained Multiobjective Optimization*, pp. 218230. Springer Nature Singapore, 2024. ISBN 9789819755813. doi: 10.1007/978-981-97-5581-3_18. URL http://dx.doi.org/10.1007/978-981-97-5581-3_18.

Ray Daniel Zimmerman, Carlos Edmundo Murillo-Sanchez, and Robert John Thomas. Matpower: Steady-state operations, planning, and analysis tools for power systems research and education. *IEEE Transactions on Power Systems*, 26(1):12–19, February 2011. ISSN 1558-0679. doi: 10.1109/tpwrs.2010.2051168. URL http://dx.doi.org/10.1109/TPWRS.2010.2051168.

A CAP AS LLM-ES FEEDBACK DESIGN

This appendix identifies the feedback requirements created by LLM-guided evolutionary search and explains how the CAP update satisfies them. LLM-ES repeatedly generates programs from scalar evaluations and accumulated search context, so its constraint handling must provide feedback with the same evolving, round-based structure.

Scalar-feedback interface. LLM-ES requires a scalar feedback signal to rank candidate programs, select parents or context, and shape subsequent program proposals. CAP supplies this interface through the recursion

$$X_i \longmapsto \boldsymbol{\lambda}_i = \mathcal{P}(X_i) \longmapsto F_{\boldsymbol{\lambda}_i} \longmapsto B_i,$$

where the batch B_i is generated under the current scalar fitness and then added to the archive. The resulting scalar fitness is therefore the constraint-aware signal used by the next LLM-ES generation.

Archive-conditioned, candidate-level frontier feedback. LLM-ES needs feedback conditioned on the evolving search history because programs discovered in earlier rounds determine the context and reference for later proposals. It also benefits from retaining candidate-level distinctions, since the raw objective quality and violation pattern of each program can inform subsequent search. CAP meets both requirements by calibrating the scalarization from the archive. The best feasible candidate currently observed defines a common frontier, and each candidate that could outrank this frontier contributes an inequality based on its own objective and positive violation vector. As the feasible frontier improves, the reference for subsequent feedback improves with it, while feasible candidates retain their raw objective ordering.

Minimum-norm penalty selection. LLM-ES receives numerical scalar fitness values rather than only the ranking they induce. A penalty that is sufficient to enforce feasibility under oracle optimization and a much larger penalty are therefore distinct feedback signals, because increasing λ changes the numerical emphasis placed on constraint violations during subsequent generation. The fixed-penalty sweeps in Appendix H.3 show that overly large penalties can make finite-round search more conservative and yield weaker best-feasible results. LLM-ES therefore benefits from a penalty that satisfies the observed frontier requirements with a controlled coefficient magnitude. Among all nonnegative penalties satisfying the candidate-level inequalities, CAP selects the one with minimum ℓ_2 norm, giving the next LLM-ES round a canonical scalarization whose penalty coefficients are no larger in ℓ_2 norm than required by the current archive constraints.

Rapid archive response. LLM-ES typically operates through short sequences of proposal rounds because LLM calls incur monetary and latency costs, while candidate evaluation can itself be time-consuming. Useful constraint feedback must therefore track archive changes at the same round granularity. CAP meets this requirement by recomputing the penalty directly from the updated frontier inequalities in one QP update. Each new batch can consequently determine the scalar feedback

used by the next LLM-ES generation, even when a small number of rounds substantially changes the available program candidates.

Together, these properties define CAP as an archive-conditioned, frontier-anchored, candidate-level, minimum-norm, and rapidly recomputed scalar-feedback mechanism for LLM-ES.

B Complete Theory and Proofs

B.1 Setup and Common Properties

This appendix proves Theorems 1 and 2 under Assumption 1. For concision, write

$$\mathbf{u}(x) := [\mathbf{v}(x)]_+.$$

Proposition B.1 (Well-defined CAP update)**.** *For every finite archive $X \subset \mathcal{X}$ containing at least one feasible point, the archive-to-penalty map $\mathcal{P}(X)$ in Definition 1 contains a unique penalty vector.*

Proof. The QP defining $\mathcal{P}(X)$ is feasible. Every feasible archive member satisfies its frontier inequality automatically, while a sufficiently large multiple of $\mathbf{1}$ satisfies the inequalities for all infeasible archive members because X is finite. The feasible set is therefore nonempty, closed, and convex. Since the objective $\frac{1}{2}\|\boldsymbol{\lambda}\|_2^2$ is strictly convex and coercive, the QP has a unique minimizer. □

Theorem B.1 (Sufficient conditions for global exact penalties)**.** *Let $\mathcal{X} \subset \mathbb{R}^n$ be compact, let $f : \mathcal{X} \to \mathbb{R}$ and $\mathbf{v} : \mathcal{X} \to \mathbb{R}^K$ be continuous, and let the feasible set $\mathcal{F} = \{x \in \mathcal{X} : \mathbf{v}(x) \le 0\}$ be nonempty. Fix a norm $\|\cdot\|_{\mathsf{x}}$ on $\mathbb{R}^n$, and suppose that there exist finite constants $L, \kappa \ge 0$ such that*

$$|f(x) - f(z)| \le L\|x - z\|_{\mathsf{x}}, \qquad \forall x, z \in \mathcal{X},$$

and

$$\mathrm{dist}_{\mathsf{x}}(x, \mathcal{F}) \le \kappa\|\mathbf{u}(x)\|_1, \qquad \forall x \in \mathcal{X}.$$

Then every $\boldsymbol{\lambda} \in \mathbb{R}_+^K$ satisfying $\lambda_j \ge L\kappa$ for all j belongs to Λ^*, so $\Lambda^* \ne \varnothing$.

Proof. Fix any $\boldsymbol{\lambda} \in \mathbb{R}_+^K$ satisfying $\lambda_j \ge L\kappa$ for all j. For any $x \in \mathcal{X}$, compactness of $\mathcal{F}$ gives a point $z_x \in \mathcal{F}$ attaining $\mathrm{dist}_{\mathsf{x}}(x, \mathcal{F})$. Since $f(z_x) \le f^*$,

$$\begin{aligned} f(x) - f^* &\le f(x) - f(z_x) \\ &\le L\|x - z_x\|_{\mathsf{x}} \\ &= L\,\mathrm{dist}_{\mathsf{x}}(x, \mathcal{F}) \\ &\le L\kappa\|\mathbf{u}(x)\|_1 \\ &\le \boldsymbol{\lambda}^\top \mathbf{u}(x). \end{aligned}$$

Therefore $F_{\boldsymbol{\lambda}}(x) \le f^*$ for every $x \in \mathcal{X}$, which is precisely $\boldsymbol{\lambda} \in \Lambda^*$. □

The theorem gives a sufficient condition for the exact-penalty assumption used below: objective growth relative to distance from the feasible set is controlled by L, while the global error bound converts that distance into aggregate constraint violation.

B.2 Single-Output Oracle

We first prove Theorem 1 for the single-output oracle case.

Lemma B.1. *If $y_i \in \mathcal{F}$, then $f(y_i) = f^*$ and $f(x_{i+1}^{\mathrm{safe}}) = f^*$.*

Proof. Let $x^* \in \arg\max_{x \in \mathcal{F}} f(x)$, which exists by compactness and continuity. Since $\mathbf{u}(x^*) = 0$,

$$F_{\boldsymbol{\lambda}_i}(x^*) = f^*.$$

Oracle optimality gives $F_{\boldsymbol{\lambda}_i}(y_i) \geq f^*$. If y_i is feasible, then $\mathbf{u}(y_i) = 0$, so $f(y_i) \geq f^*$. Feasibility also gives $f(y_i) \leq f^*$, proving $f(y_i) = f^*$. The archive update then yields $f(x_{i+1}^{\text{safe}}) = f^*$. □

Lemma B.2. *Suppose that, for some i_0 and $c < f^*$, $f(x_i^{\text{safe}}) = c$ for every $i \geq i_0$. Then*

$$\|\boldsymbol{\lambda}_i\|_2 \to \infty.$$

Proof. For $i \geq i_0$, define

$$C_i := \left\{ \boldsymbol{\lambda} \in \mathbb{R}_+^K : \boldsymbol{\lambda}^\top \mathbf{u}(x) \geq f(x) - c,\ \forall x \in X_i \right\}.$$

Then $\boldsymbol{\lambda}_i$ is the unique Euclidean projection of the origin onto C_i, and $C_{i+1} \subseteq C_i$. By Lemma B.1, $y_i \notin \mathcal{F}$ on the assumed suboptimal plateau. Oracle optimality therefore gives

$$F_{\boldsymbol{\lambda}_i}(y_i) \geq F_{\boldsymbol{\lambda}_i}(x^*) = f^* > c,$$

which is equivalent to

$$\boldsymbol{\lambda}_i^\top \mathbf{u}(y_i) < f(y_i) - c.$$

Thus $\boldsymbol{\lambda}_i \notin C_{i+1}$. Since the projection onto C_i is unique, this strict exclusion implies

$$\|\boldsymbol{\lambda}_{i+1}\|_2 > \|\boldsymbol{\lambda}_i\|_2. \tag{B.1}$$

Assume for contradiction that $\{\|\boldsymbol{\lambda}_i\|_2\}$ is bounded. Then Equation B.1 implies that this strictly increasing sequence converges. Let

$$C_\infty := \bigcap_{i \geq i_0} C_i.$$

Every convergent subsequence of $\{\boldsymbol{\lambda}_i\}_{i \geq i_0}$ has a limit in C_∞: for each fixed j, all sufficiently late members belong to $C_i \subseteq C_j$, and C_j is closed. Moreover, for every $\boldsymbol{\lambda} \in C_\infty$, projection optimality gives

$$\|\boldsymbol{\lambda}_i\|_2 \leq \|\boldsymbol{\lambda}\|_2.$$

Hence every accumulation point is the minimum-norm point of C_∞, which is unique. Since the sequence is bounded in $\mathbb{R}^K$, every subsequence has a convergent subsequence. The uniqueness of the accumulation point therefore implies that the full sequence converges to some $\bar{\boldsymbol{\lambda}} \in C_\infty$.

By compactness of $\mathcal{X}$, there exist a subsequence $\{y_{i_r}\}$ and a point $\bar{y} \in \mathcal{X}$ such that $y_{i_r} \to \bar{y}$. Since $y_{i_r} \in X_{i_r+1}$, the next master problem gives

$$\boldsymbol{\lambda}_{i_r+1}^\top \mathbf{u}(y_{i_r}) \geq f(y_{i_r}) - c.$$

Taking limits yields

$$F_{\bar{\boldsymbol{\lambda}}}(\bar{y}) \leq c.$$

On the other hand, let x^* be a constrained global optimizer. Since y_{i_r} maximizes $F_{\boldsymbol{\lambda}_{i_r}}$ and x^* is feasible, for every r,

$$F_{\boldsymbol{\lambda}_{i_r}}(y_{i_r}) \geq F_{\boldsymbol{\lambda}_{i_r}}(x^*) = f^*.$$

As $\boldsymbol{\lambda}_{i_r} \to \bar{\boldsymbol{\lambda}}$ and $y_{i_r} \to \bar{y}$, continuity of f and $\mathbf{u}$ gives

$$F_{\bar{\boldsymbol{\lambda}}}(\bar{y}) \geq f^* > c,$$

a contradiction. Thus the penalty sequence is unbounded. □

Proof of Theorem 1. Suppose for contradiction that no finite archive X_i contains a globally optimal solution. In particular, $f(x_0^{\text{safe}}) < f^*$, and no oracle output y_i is feasible by Lemma B.1. Hence $f(x_i^{\text{safe}}) = f(x_0^{\text{safe}}) < f^*$ for every i, so Lemma B.2 gives $\|\boldsymbol{\lambda}_i\|_2 \to \infty$. Choose any $\boldsymbol{\lambda}^* \in \Lambda^*$. By asymptotic direction coverage, there is a finite iteration i such that

$$\min_{j=1,\dots,K} (\boldsymbol{\lambda}_i)_j > \max_{j=1,\dots,K} (\boldsymbol{\lambda}^*)_j.$$

Hence $\boldsymbol{\lambda}_i > \boldsymbol{\lambda}^*$ componentwise. For every infeasible x, $\mathbf{u}(x) \neq 0$, and therefore

$$F_{\boldsymbol{\lambda}_i}(x) = F_{\boldsymbol{\lambda}^*}(x) - (\boldsymbol{\lambda}_i - \boldsymbol{\lambda}^*)^\top \mathbf{u}(x) < f^*.$$

Every constrained global optimizer has penalized value f^*, so every oracle maximizer must be feasible. Lemma B.1 then implies that X_{i+1} contains a globally optimal solution, a contradiction. □

B.3 Batch-Oracle Asymptotic Optimality

Proof of Theorem 2. Each archive remains finite by induction. The sequence $\{f(x_i^{\text{safe}})\}$ is nondecreasing and bounded above by f^*; let $\bar{f} := \lim_{i\to\infty} f(x_i^{\text{safe}}) \leq f^*$. Suppose for contradiction that $\bar{f} < f^*$, and define

$$X_\infty := \bigcup_{i \geq 0} X_i$$

and

$$C_\infty(\bar{f}) := \left\{ \boldsymbol{\lambda} \in \mathbb{R}_+^K : \boldsymbol{\lambda}^\top \mathbf{u}(x) \geq f(x) - \bar{f},\ \forall x \in X_\infty \right\}. \tag{B.2}$$

We first show that $\{\boldsymbol{\lambda}_i\}$ is unbounded. Otherwise, take a convergent subsequence $\boldsymbol{\lambda}_{i_r} \to \bar{\boldsymbol{\lambda}}$. For every fixed j, sufficiently large i_r satisfy $X_j \subseteq X_{i_r}$ and $f(x_{i_r}^{\text{safe}}) \leq \bar{f}$. Thus the master constraints at iteration i_r, restricted to X_j, are at least as strong as the corresponding constraints in Equation B.2. Closedness gives

$$\bar{\boldsymbol{\lambda}} \in C_\infty(\bar{f}). \tag{B.3}$$

For each r, select

$$y_{i_r} \in B_{i_r} \cap \arg\max_{x \in \mathcal{X}} F_{\boldsymbol{\lambda}_{i_r}}(x).$$

By compactness, pass to a further subsequence with $y_{i_r} \to \bar{y}$. Because $y_{i_r} \in X_\infty$, Equation B.3 implies

$$\bar{\boldsymbol{\lambda}}^\top \mathbf{u}(y_{i_r}) \geq f(y_{i_r}) - \bar{f}.$$

Taking limits gives

$$F_{\bar{\boldsymbol{\lambda}}}(\bar{y}) \leq \bar{f}. \tag{B.4}$$

For a constrained global optimizer x^*, oracle optimality gives

$$F_{\boldsymbol{\lambda}_{i_r}}(y_{i_r}) \geq F_{\boldsymbol{\lambda}_{i_r}}(x^*) = f^*.$$

Taking limits contradicts Equation B.4:

$$F_{\bar{\boldsymbol{\lambda}}}(\bar{y}) \geq f^* > \bar{f}.$$

Thus $\{\boldsymbol{\lambda}_i\}$ is unbounded. Select $\boldsymbol{\lambda}^* \in \Lambda^*$. By asymptotic direction coverage, there is a finite iteration i such that

$$\min_{j=1,\ldots,K}(\boldsymbol{\lambda}_i)_j > \max_{j=1,\ldots,K}(\boldsymbol{\lambda}^*)_j,$$

and hence $\boldsymbol{\lambda}_i > \boldsymbol{\lambda}^*$ componentwise. The same strict exact-penalty argument used in the proof of Theorem 1 shows that every global maximizer of $F_{\boldsymbol{\lambda}_i}$ is feasible and constrained-optimal. The batch assumption then gives $f(x_{i+1}^{\text{safe}}) = f^*$, contradicting $\bar{f} < f^*$. Therefore

$$\lim_{i\to\infty} f(x_i^{\text{safe}}) = f^*.$$

□

B.4 Remarks on the Two Results

Theorem 1 has a finite-time conclusion because, until a feasible oracle output occurs, the feasible archive value remains fixed. The accumulated master constraints are then nested, and Lemma B.2 shows that the minimum-norm penalty diverges on any suboptimal plateau.

Theorem 1 gives the stronger finite-time conclusion, but its single-output setting describes a particularly abrupt search event: the feasible archive can remain at one suboptimal value until the oracle suddenly returns a globally optimal feasible point. Such a jump is unlikely to characterize evolutionary search, where newly generated candidates are typically local variations of existing ones and feasible improvements are expected to accumulate gradually. Theorem 2 captures this regime by allowing a batch to contain feasible candidates that improve the archive even when no batch member is yet a constrained global optimizer. Its asymptotic conclusion is weaker, but it better matches the incremental progress produced by evolutionary search. Technically, these additional batch members can also make the master constraints relax between iterations, so the nested-set argument used for finite-time discovery no longer applies; the limiting-system argument instead rules out convergence of the feasible archive value to any $\bar{f} < f^*$.

The exact-penalty and asymptotic-direction-coverage conditions play distinct roles. The existence of $\boldsymbol{\lambda}^* \in \Lambda^*$ provides a penalty that globally suppresses every infeasible point at f^*. Asymptotic direction coverage ensures that an unbounded penalty sequence contains an iterate that strictly dominates $\boldsymbol{\lambda}^*$ componentwise. At that iterate, every infeasible point has penalized value strictly below f^*, which yields feasibility and constrained optimality for every penalized global maximizer.

The proofs also show that the archive need not grow monotonically. Define $\mathrm{ND}(X_i)$ as the subset of archive points that are non-dominated under the ordering "larger f and smaller $\mathbf{u}$ are better." Then the following sufficiency result follows directly.

Corollary B.1 (Pareto-frontier sufficiency)**.** *At any iteration i, replacing X_i by $\mathrm{ND}(X_i)$ leaves both the best feasible archive value and the CAP master-QP solution unchanged:*

$$\max_{x\in X_i\cap\mathcal{F}} f(x) = \max_{x\in \mathrm{ND}(X_i)\cap\mathcal{F}} f(x), \qquad \boldsymbol{\lambda}_i(X_i) = \boldsymbol{\lambda}_i(\mathrm{ND}(X_i)).$$

Indeed, every point removed from X_i is dominated by a retained point. The retained point preserves or improves the feasible objective value and its master-QP inequality implies the inequality of the dominated point for every nonnegative penalty vector. Repeated application of this observation removes all dominated points without changing either quantity.

C LLM-GUIDED EVOLUTIONARY SEARCH WORKFLOW

This appendix describes a typical LLM-guided evolutionary search (LLM-ES) workflow for unconstrained program optimization.

C.1 POPULATION AND OBJECTIVE

Let $\mathcal{X}$ denote the space of candidate programs and let $f : \mathcal{X} \to \mathbb{R}$ be the objective to maximize. Evaluating a program $x \in \mathcal{X}$ executes it on the task evaluator and returns the scalar fitness $f(x)$.

At LLM-ES iteration t, let $X_t \subset \mathcal{X}$ denote the finite population available for subsequent generation and selection. The population need not contain every program previously evaluated: a bounded implementation may remove candidates while preserving selected elites, diverse candidates, or other useful search history.

C.2 A TYPICAL LLM-ES ITERATION

A typical LLM-ES iteration consists of candidate selection, LLM-based mutation, evaluation, and population update. First, a selection operator chooses one or more programs from the population:

$$C_t = \mathcal{S}(X_t; f).$$

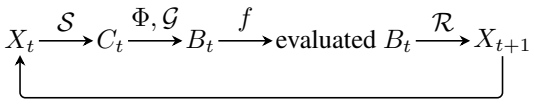
The selected programs and their evaluation feedback form the mutation context C_t. A prompt builder Φ combines this context with the task instructions, and an LLM-based mutation operator $\mathcal{G}$ produces a finite candidate batch:

$$Z_t = \Phi(C_t), \qquad B_t = \mathcal{G}(Z_t) \subset \mathcal{X}.$$

Each candidate $x \in B_t$ is then executed by the task evaluator to obtain $f(x)$. Finally, an update operator incorporates the evaluated batch into the population:

$$X_{t+1} = \mathcal{R}(X_t, B_t; f).$$

The update may retain, replace, rank, or remove candidates according to the particular LLM-ES design. The four operations therefore form the composite transition



Unlike a conventional numerical mutation operator, $\mathcal{G}$ proposes semantic modifications in program space by conditioning an LLM on the selected context. Nevertheless, the surrounding evolutionary logic remains the same: evaluated search history determines which programs guide generation and which generated programs remain available to future iterations.

C.3 ISLAND-STRUCTURED SEARCH

An island-structured population consists of S persistent local subpopulations, called islands,

$$\mathbb{X}_t = \big(X_t^{(1)}, \ldots, X_t^{(S)}\big),$$

where $X_t^{(s)}$ is the population of island s. Each island performs the same basic LLM-ES loop but selects parents primarily from its own population and applies its own local update. If iteration t operates on island s, then

$$C_t^{(s)} = \mathcal{S}^{(s)}(X_t^{(s)}; f), \qquad B_t^{(s)} = \mathcal{G}\Big(\Phi(C_t^{(s)})\Big), \qquad X_{t+1}^{(s)} = \mathcal{R}^{(s)}(X_t^{(s)}, B_t^{(s)}; f),$$

while the other island populations remain unchanged at that iteration. Because parent selection and retention are mostly local, different islands can preserve distinct candidate lineages and explore different regions of program space.

Islands may evolve independently or exchange information during search. One common mechanism is periodic migration, which transfers selected programs between island populations and allows a useful discovery from one island to influence another without collapsing all islands into a single population. Regardless of whether information exchange is used, an island is not merely a temporary generation batch: it is a persistent subpopulation with its own local evolutionary trajectory.

D Complete CAP-Evolve Algorithm and Implementation Details

This appendix records the implementation details shared by the main experiments. Unless stated otherwise, all tasks use the same CAP-Evolve implementation and differ only in their evaluator, prompt, and constraint dimension.

D.1 Relation to Idealized CAP

CAP-Evolve instantiates CAP in executable program space by using AdaEvolve as the inner-search operator. It preserves CAP's defining frontier update: the best feasible archive candidate provides the reference value, observed infeasible counterexamples impose frontier inequalities, and the minimum-norm penalty determines the scalar fitness used to guide subsequent search.

Moving from the oracle setting to finite-budget LLM-ES requires additional implementation choices. CAP-Evolve uses bounded island-local archives, active-count-normalized violations, periodic penalty updates, and alpha relaxation. Table D.1 summarizes these differences and their practical rationale. Accordingly, the theoretical results characterize the idealized mechanism under oracle assumptions, while the experiments evaluate its practical CAP-Evolve instantiation.

Table D.1: Differences between CAP and CAP-Evolve.

Aspect	Idealized CAP	CAP-Evolve	Practical rationale
Search space	Compact continuous search space	Executable-program search space	Models the penalty mechanism in a tractable mathematical setting while the application searches over code.
Inner search	Global-maximizer oracle	Finite-budget AdaEvolve search	Replaces the unavailable oracle with an operational LLM-ES procedure.
Archive/penalty scope	Global archive and penalty	Island-local archives and penalties	Preserves AdaEvolve's distinct island search dynamics.
Violation signal	Raw positive violations $\mathbf{u}(x)$	Normalized positive violations $\tilde{\mathbf{u}}(x)$	Reduces sensitivity to the number of similarly scaled active constraints.
Frontier update	Every iteration using all archive rows	Every h iterations using alpha-relaxed eligible rows	Uses weaker early penalties to favor exploration, then restores the full retained frontier for exploitation.
Direction coverage	Asymptotic direction coverage assumed	Asymptotic direction coverage not enforced	Focuses finite-budget updates on suppressing retained counterexamples rather than enforcing the asymptotic condition.

D.2 Common Configuration

CAP-Evolve is implemented by extending AdaEvolve's scalar-fitness interface, and each run uses a budget of 100 evolutionary iterations. Table D.2 reports the common AdaEvolve and CAP-Evolve configuration, Table D.3 gives the shared constraint-handling parameters, and Table D.4 lists the task-specific initial alpha relaxation fractions. Other task-specific changes are limited to the evaluator, prompt, and constraint dimension.

Table D.2: Common configuration.

Group	Parameter	Value
–	diff_based_generation	true
	max_solution_length	60000
evaluator	timeout	660
database	num_islands	2
	decay	0.9
	intensity_min	0.15
	intensity_max	0.5
	migration_interval	15
	migration_count	5
	use_adaptive_search	true
	use_ucb_selection	true
	use_migration	true
	use_unified_archive	true
	use_paradigm_breakthrough	true
	use_dynamic_islands	true

Table D.3: Constraint-handling configuration.

Parameter	Value
lambda_init	5.0
lambda_update_interval	5
alpha_relaxation_initial_fraction α_0	task-specific
alpha_relaxation_end_iteration	80
active_count_exponent	0.5

Table D.4: Task-specific alpha fractions.

Task	α_0
Circle Packing	0.5
Spectral Shape Optimization	0.75
Trajectory Optimization	0.5
Tokamak Control	0.25

D.3 LLM CONFIGURATION

All experiments use a locally deployed OpenAI `gpt-oss-20b` model. Candidate generation uses the settings in Table D.5. Parameters not listed in the table are left unset by the search framework and follow the serving backend defaults.

Table D.5: Shared LLM configuration.

Parameter	Value
model	`gpt-oss-20b`
temperature	0.7
top_p	unset
reasoning_effort	unset
max_tokens	16384
request_timeout	600 s
request_retries	3
retry_delay	5 s

Candidate-generation requests are issued sequentially within each run. The generated programs and task evaluators execute on CPUs, and candidate evaluations are not run concurrently.

D.4 Practical Penalty Update

AdaEvolve maintains multiple interacting islands. Each island maintains its own penalty vector, archive, and fitness values. A new candidate is scored with the current penalty of its assigned island; after an island penalty update, all retained candidates on that island are rescored before subsequent selection and parent sampling.

At an AdaEvolve iteration t at which a penalty update is triggered, let x^{safe} be the feasible archive member with the largest raw objective. For each infeasible archive member y, define

$$\mathbf{q}_y = \widetilde{\mathbf{u}}(y), \qquad b_y = f(y) - f(x^{\mathrm{safe}}).$$

Candidates with $b_y \le 0$ or $\mathbf{q}_y = \mathbf{0}$ do not create an active frontier row. The frontier QP is solved over the remaining rows.

Rows are ranked by the difficulty score

$$d_y = \frac{b_y}{\|\mathbf{q}_y\|_2},$$

with b_y as the tie breaker. At AdaEvolve iteration t, the relaxation coefficient starts from α_0 and decreases linearly to zero by iteration 80. The relaxed update discards the highest-difficulty rows according to $\lfloor \alpha_t N \rfloor$, where N is the number of candidate rows, while retaining at least one row whenever $N > 0$. Thus, after the schedule reaches zero, all eligible frontier rows are included in the frontier QP.

D.5 Handling Infeasible Initializations

Definition 1 assumes that the initial archive contains a feasible candidate, which provides the safe frontier used by the QP. When this assumption does not hold empirically, one possible treatment is to adjust each penalty component using the fraction of retained candidates that satisfy the corresponding individual constraint: components below a lower threshold are increased and those above an upper threshold are decreased. Once a feasible candidate is available, the regular frontier-QP update resumes.

The reported experiments use feasible initial solution sets, so this situation does not arise.

E Tessema–Yen Adaptive-Penalty Baseline

We introduce the classical self-adaptive penalty method from evolutionary computation, due to Tessema & Yen (2006), in the maximization problem defined in Section 3.1. The method constructs a population-dependent scalar fitness without learning an explicit penalty vector. The following specification expresses its normalized constraint aggregation in terms of our K-dimensional signed evaluator output.

E.1 Scalarization in Our Setting

Let $X \subset \mathcal{X}$ be the current finite retained population used for selection. Define $f_X^{\max} = \max_{z \in X} f(z)$ and $f_X^{\min} = \min_{z \in X} f(z)$. Because our objective is maximized, we use the normalized objective cost

$$\widehat{f}_X(x) = \begin{cases} \dfrac{f_X^{\max} - f(x)}{f_X^{\max} - f_X^{\min}}, & f_X^{\max} > f_X^{\min}, \\ 0, & f_X^{\max} = f_X^{\min}. \end{cases}$$

For each constraint k, let $u_k(x) = [v_k(x)]_+$ and $M_{X,k} = \max_{z \in X} u_k(z)$. We normalize each violation component and then average across components:

$$\widehat{u}_{X,k}(x) = \begin{cases} u_k(x)/M_{X,k}, & M_{X,k} > 0, \\ 0, & M_{X,k} = 0, \end{cases} \qquad \widehat{v}_X(x) = \frac{1}{K}\sum_{k=1}^{K} \widehat{u}_{X,k}(x).$$

The feasible fraction of the population is

$$\rho_X = \frac{|\{x \in X : \mathbf{v}(x) \le \mathbf{0}\}|}{|X|}.$$

Following the original method, define the distance value

$$d_X(x) = \begin{cases} \widehat{v}_X(x), & \rho_X = 0, \\ \sqrt{\widehat{f}_X(x)^2 + \widehat{v}_X(x)^2}, & \rho_X > 0, \end{cases}$$

and the additional penalty

$$p_X(x) = (1 - \rho_X)a_X(x) + \rho_X b_X(x),$$

where

$$a_X(x) = \begin{cases} 0, & \rho_X = 0, \\ \widehat{v}_X(x), & \rho_X > 0, \end{cases} \qquad b_X(x) = \begin{cases} 0, & \mathbf{v}(x) \le \mathbf{0}, \\ \widehat{f}_X(x), & \text{otherwise.} \end{cases}$$

The original method minimizes $d_X(x) + p_X(x)$. To match AdaEvolve's higher-is-better scalar-fitness interface, our implementation instead maximizes $F_X^{\mathrm{TY}}(x) = -\big(d_X(x) + p_X(x)\big)$.

Thus, when X contains no feasible candidate, candidates are ranked only by aggregate violation. When all candidates are feasible, their ordering agrees with the raw maximization objective. Between these cases, the feasible fraction controls the balance between normalized objective quality and violation.

E.2 AdaEvolve Integration

Each AdaEvolve island applies the preceding construction independently to its current retained population. Whenever an island population changes, its population statistics and F_X^{TY} values are recomputed for all retained candidates before subsequent parent selection. A newly evaluated program therefore updates the scalar scores used by its island but does not alter those of other islands. All remaining search, evaluator, prompting, budget, and final-candidate selection settings are shared with the other methods in the main comparison.

F Benchmark Definitions, Sources, and Evaluator Details

All benchmark evaluators return a raw scalar objective and a signed violation vector. A candidate is feasible precisely when every returned violation is nonpositive. The evaluation procedure uses fixed instances and seed sets.

F.1 Circle Packing

Each candidate specifies 26 circle centers and radii in the unit square. The evaluator recomputes the radius sum and uses

$$f(x) = \frac{\sum_{i=1}^{26} r_i}{2.635},$$

where 2.635 is the reference target value. A packing must have nonnegative radii, lie within the unit square, and contain no overlapping circle pair. A tolerance of 10^{-6} is subtracted from each signed geometric residual.

We use three equivalent representations of the same geometric feasible set. The one-dimensional representation returns the maximum residual over all geometric conditions. The three-dimensional representation separately returns the maximum boundary, overlap, and negative-radius residuals. The 26-dimensional representation returns one maximum residual per circle, combining that circle's boundary, radius, and pairwise-overlap residuals.

Circle packing is a standard optimization problem (Blind & Ament, 2000) and is included in the AdaEvolve evaluation suite (Novikov et al., 2025). The task formulation and its three constraint representations are specified by our evaluators.

F.2 Spectral Shape Optimization

Each candidate specifies a two-dimensional domain. The evaluator computes Dirichlet Laplacian eigenvalues, namely the eigenvalues of the Laplacian operator with zero boundary conditions on the domain. The target is to minimize the fifth eigenvalue under a unit-area constraint; it is expressed as the maximization objective

$$f(x) = \frac{82.41195175097458}{\lambda_5(x)}, \qquad v(x) = \mathrm{area}(x) - 1.$$

The reference numerator is the fifth eigenvalue of a unit-area disk under the same finite-difference backend.

Spectral shape optimization is an established mathematical problem, including minimization of the k-th Dirichlet-Laplacian eigenvalue (Bucur, 2012). Our evaluator uses the fifth eigenvalue, normalized by a fixed unit-area disk reference.

F.3 Multi-Agent Trajectory Optimization

Each candidate specifies open-loop trajectories for 16 agents using degree-10 Bernstein curves with 11 control coefficients per coordinate. The evaluation suite contains 100 fixed environments generated from the same eight-obstacle scenario family, indexed by seeds 0 through 99. In each environment, the sixteen agents move between prescribed endpoints while avoiding static ellipsoidal obstacles and one another. The evaluator samples each trajectory at 100 points on $[0, 10]$. Per-environment quality is the negative weighted squared acceleration, normalized by a fixed reference; the x-, y-, and z-acceleration weights are 100, 100, and 3000. The reported objective is the mean normalized quality across the 100 fixed environments.

The five-dimensional constraint representation comprises the maximum signed residuals for endpoint position, endpoint velocity, endpoint acceleration, robot–robot collision, and robot–obstacle collision. Collision residuals use axis-aligned ellipsoidal clearance: for relative displacement $\mathbf{d}$ and exclusion axes $\mathbf{a}$,

$$v_{\mathrm{collision}} = 1 - \sum_{\ell=1}^{3} (d_\ell / a_\ell)^2.$$

Each collision component first takes the maximum over all relevant pairs and sampled times, then the maximum over seeds. The single-constraint version returns the maximum of the five final components. This adapts the convex-approximation trajectory benchmark of Rastgar et al. (2021) to our executable-program interface, fixed seed set, and signed-violation protocol.

F.4 Tokamak Control

Each candidate specifies a feedback policy mapping state and history to the two actuator commands S_{DT} and S_{S}. It is evaluated on 100 fixed episodes with seeds 0 through 99, using time step 0.1 and

a horizon of 20 seconds. The raw maximization objective is the negative mean relative D–T density tracking error,

$$f(x) = -\frac{1}{100} \sum_{\text{episodes}} \text{mean}_t \frac{|n_{\text{DT},t} - \bar{n}_{\text{DT},t}|}{|\bar{n}_{\text{DT},t}|}.$$

The single signed constraint is the worst relative safety residual over all episodes and time steps:

$$v(x) = \max_{\text{episodes},t} \frac{n_{e,t} - c_{S,t} n_{gW,t}}{\max\{|n_{e,t}|, |c_{S,t} n_{gW,t}|\}}.$$

Thus $v(x) \leq 0$ is equivalent to satisfying the electron-density safety condition throughout all fixed episodes. The safety condition and density-regulation setting follow the disruption-avoidance controller formulation of Paruchuri et al. (2025); we adapt it to a program-construction evaluator with fixed episodes and signed residuals.

F.5 Shared Initial Programs

All methods and independent runs start from the same task-specific initial program. Each initial program produces a feasible candidate under its corresponding evaluator.

- **Circle Packing.** The initial layout places one circle near the center, eight circles on an inner ring, and sixteen circles on an outer ring. Circle radii are then chosen from the available distance to the square boundary and to neighboring centers.
- **Spectral Shape Optimization.** The initial domain is a disk centered at the origin with radius 0.55, giving an area below the unit-area limit.
- **Multi-Agent Trajectory Optimization.** The initial program directly reproduces the initialization from the source trajectory-planning work (Rastgar et al., 2021), with only the interface adaptation needed by our evaluator.
- **Tokamak Control.** The initial controller applies zero deuterium–tritium fueling and zero helium injection at every step. It therefore provides a simple safe feedback-policy starting point.

G Complete Main-Experiment Results

This appendix gives the numerical values underlying Table 1. Each entry is the task-native metric of the best feasible retained program from one independent run. We report the five run values, their mean and sample standard deviation (computed across the five independent runs with denominator $n - 1$), and the number of runs yielding a feasible final candidate. The preferred direction is higher for Circle Packing and Spectral Shape Optimization and lower for Trajectory Optimization and Tokamak Control. The method labels agree with Table 1. In the K column, "–" denotes a row that covers all available constraint representations for that task.

G.1 Circle Packing

Table G.1: Circle Packing results.

K	Method	Run 1	Run 2	Run 3	Run 4	Run 5	Mean ± s.d.	Feasible runs
–	AdaEvolve (hard rejection)	0.99973	0.94581	0.99413	0.99410	0.99654	$0.98606_{\pm 0.02262}$	5 / 5
1	AdaEvolve (best fixed penalty)	0.99167	1.00034	0.99697	0.99779	0.99312	$0.99598_{\pm 0.00354}$	5 / 5
1	AdaEvolve (T–Y penalty)	0.997345	0.958582	0.979229	0.990334	0.997656	$0.98463_{\pm 0.01637}$	5 / 5
1	CAP-Evolve	0.99611	0.99750	0.99899	0.99842	0.99132	$0.99647_{\pm 0.00308}$	5 / 5
3	AdaEvolve (T–Y penalty)	0.993894	0.998759	0.992644	0.990316	1.000373	$0.99520_{\pm 0.00423}$	5 / 5

K	Method	Run 1	Run 2	Run 3	Run 4	Run 5	Mean ± s.d.	Feasible runs
3	CAP-Evolve	0.996329	0.996696	0.998991	0.987789	0.998269	$0.99561_{\pm 0.00451}$	5 / 5
26	AdaEvolve (T–Y penalty)	0.991739	0.997925	0.999254	0.964486	0.997043	$0.99009_{\pm 0.01459}$	5 / 5
26	CAP-Evolve	0.99694	0.99861	0.99859	0.99899	0.99925	$0.99848_{\pm 0.00090}$	5 / 5

G.2 Spectral Shape Optimization

Table G.2: Spectral Shape Optimization results.

Method	Run 1	Run 2	Run 3	Run 4	Run 5	Mean ± s.d.	Feasible runs
AdaEvolve (hard rejection)	1.02982	1.00882	1.00037	1.00870	1.00634	$1.01080_{\pm 0.01120}$	5 / 5
AdaEvolve (best fixed penalty)	1.01977	1.01183	1.01111	1.01157	1.05586	$1.02200_{\pm 0.01920}$	5 / 5
AdaEvolve (T–Y penalty)	1.009050	1.011360	1.008504	1.008526	1.019167	$1.01130_{\pm 0.00450}$	5 / 5
CAP-Evolve	1.00006	1.00904	1.00911	1.03906	1.05809	$1.02310_{\pm 0.02450}$	5 / 5

G.3 Trajectory Optimization

Table G.3: Trajectory Optimization results.

K	Method	Run 1	Run 2	Run 3	Run 4	Run 5	Mean ± s.d.	Feasible runs
–	AdaEvolve (hard rejection)	1.36077	1.35946	1.35993	1.36077	1.34894	$1.35797_{\pm 0.00508}$	5 / 5
1	AdaEvolve (best fixed penalty)	1.36077	1.36070	1.35919	1.35474	1.36077	$1.35924_{\pm 0.00260}$	5 / 5
1	AdaEvolve (T–Y penalty)	1.354478	1.360770	1.360772	1.360772	1.360772	$1.35951_{\pm 0.00281}$	5 / 5
1	CAP-Evolve	1.35922	1.36077	1.36077	1.35518	1.36077	$1.35934_{\pm 0.00242}$	5 / 5
5	AdaEvolve (T–Y penalty)	1.356369	1.360772	1.360772	1.352136	1.360772	$1.35816_{\pm 0.00387}$	5 / 5
5	CAP-Evolve	1.34914	1.35944	1.35866	1.35944	1.34801	$1.35494_{\pm 0.00583}$	5 / 5

G.4 Tokamak Control

Table G.4: Tokamak Control results.

Method	Run 1	Run 2	Run 3	Run 4	Run 5	Mean ± s.d.	Feasible runs
AdaEvolve (hard rejection)	0.06155	0.07969	0.06550	0.06160	0.06446	$0.06656_{\pm 0.00754}$	5 / 5
AdaEvolve (best fixed penalty)	0.06468	0.06331	0.06481	0.06542	0.06808	$0.06526_{\pm 0.00175}$	5 / 5
AdaEvolve (T–Y penalty)	0.066212	0.062023	0.126498	0.061657	0.065940	$0.07647_{\pm 0.02805}$	5 / 5
CAP-Evolve	0.06451	0.06202	0.06540	0.07051	0.06341	$0.06517_{\pm 0.00324}$	5 / 5

The selected fixed-penalty values and their full sweeps are reported in Appendix H, together with supplementary component results and alpha sensitivity.

G.5 Reference Problem Metrics

For comparability with reports that use the untransformed problem quantities, Table G.5 gives a reference metric for the same final feasible programs used in Tables G.1–G.4. These values are not used for search, final-candidate selection, or the main performance comparisons. Circle Packing reports the sum of radii; Spectral Shape Optimization reports the fifth Dirichlet eigenvalue; Trajectory Optimization reports the unnormalized weighted squared-acceleration objective; and Tokamak Control reports mean cumulative absolute D–T density tracking error over the fixed episodes.

Table G.5: Reference problem metrics.

Task	K	Method	Reference metric	Mean ± s.d.
Circle Packing	–	AdaEvolve (hard rejection)	$\sum_i r_i \uparrow$	$2.59828_{\pm 0.05960}$
Circle Packing	1	AdaEvolve (best fixed penalty)	$\sum_i r_i \uparrow$	$2.62440_{\pm 0.00932}$
Circle Packing	1	AdaEvolve (T–Y penalty)	$\sum_i r_i \uparrow$	$2.59450_{\pm 0.04313}$
Circle Packing	1	CAP-Evolve	$\sum_i r_i \uparrow$	$2.62570_{\pm 0.00810}$
Circle Packing	3	AdaEvolve (T–Y penalty)	$\sum_i r_i \uparrow$	$2.62235_{\pm 0.01114}$
Circle Packing	3	CAP-Evolve	$\sum_i r_i \uparrow$	$2.62345_{\pm 0.01188}$
Circle Packing	26	AdaEvolve (T–Y penalty)	$\sum_i r_i \uparrow$	$2.60889_{\pm 0.03846}$
Circle Packing	26	CAP-Evolve	$\sum_i r_i \uparrow$	$2.63099_{\pm 0.00238}$
Spectral Shape Optimization	1	AdaEvolve (hard rejection)	$\lambda_5 \downarrow$	$81.53856_{\pm 0.89091}$
Spectral Shape Optimization	1	AdaEvolve (best fixed penalty)	$\lambda_5 \downarrow$	$80.65817_{\pm 1.48476}$
Spectral Shape Optimization	1	AdaEvolve (T–Y penalty)	$\lambda_5 \downarrow$	$81.49073_{\pm 0.36401}$
Spectral Shape Optimization	1	CAP-Evolve	$\lambda_5 \downarrow$	$80.59004_{\pm 1.90860}$
Trajectory Optimization	–	AdaEvolve (hard rejection)	Weighted squared acceleration ↓	$24779.52000_{\pm 92.76760}$
Trajectory Optimization	1	AdaEvolve (best fixed penalty)	Weighted squared acceleration ↓	$24802.54700_{\pm 47.44150}$
Trajectory Optimization	1	AdaEvolve (T–Y penalty)	Weighted squared acceleration ↓	$24807.59500_{\pm 51.36160}$
Trajectory Optimization	1	CAP-Evolve	Weighted squared acceleration ↓	$24804.48960_{\pm 44.21750}$
Trajectory Optimization	5	AdaEvolve (T–Y penalty)	Weighted squared acceleration ↓	$24782.98740_{\pm 70.64940}$
Trajectory Optimization	5	CAP-Evolve	Weighted squared acceleration ↓	$24724.13610_{\pm 106.43660}$
Tokamak Control	1	AdaEvolve (hard rejection)	Cumulative absolute D–T tracking error ↓	$1.39197 \times 10^{21}{}_{\pm 1.25322 \times 10^{20}}$
Tokamak Control	1	AdaEvolve (best fixed penalty)	Cumulative absolute D–T tracking error ↓	$1.37233 \times 10^{21}{}_{\pm 3.60910 \times 10^{19}}$
Tokamak Control	1	AdaEvolve (T–Y penalty)	Cumulative absolute D–T tracking error ↓	$1.61208 \times 10^{21}{}_{\pm 5.77280 \times 10^{20}}$
Tokamak Control	1	CAP-Evolve	Cumulative absolute D–T tracking error ↓	$1.37302 \times 10^{21}{}_{\pm 6.21206 \times 10^{19}}$

H Supplementary Component Results, Alpha Sensitivity, and Fixed-Penalty Sweeps

This appendix supplements the aggregate component comparisons in Table 2 and documents the two scalar-penalty diagnostics used in the study. It does not introduce additional primary comparisons.

H.1 Run-Level Circle Packing Component Results

Table H.1 expands Table 2 into the five final best-feasible values from the independent runs of every Circle Packing component configuration. It contains the complete alpha-relaxation and penalty-scope comparisons at $K = 1, 3, 26$, and the complete normalization comparison at $K = 3, 26$. Higher target ratios are better. The default alpha setting is $\alpha_0 = 0.5$; *None* denotes no active-count normalization.

Table H.1: Circle Packing component results.

K	Normalization	α_0	Penalty scope	Run 1	Run 2	Run 3	Run 4	Run 5	Mean ± s.d.	Feasible runs
1	None	0.5	Island	0.99611	0.99750	0.99899	0.99842	0.99132	$0.99647_{\pm 0.00308}$	5 / 5
1	Sqrt	0.5	Global	0.998418	0.978647	0.996823	0.989975	0.992046	$0.99118_{\pm 0.00780}$	5 / 5
1	None	0	Island	0.99832	0.99120	0.99661	0.99384	0.99837	$0.99567_{\pm 0.00310}$	5 / 5
3	None	0.5	Island	0.99302	1.00037	0.99604	0.99807	0.99671	$0.99684_{\pm 0.00270}$	5 / 5
3	Mean	0.5	Island	0.99611	1.00037	0.98857	0.99840	0.99085	$0.99486_{\pm 0.00500}$	5 / 5
3	Sqrt	0.5	Island	0.996329	0.996696	0.998991	0.987789	0.998269	$0.99561_{\pm 0.00451}$	5 / 5
3	Sqrt	0.5	Global	1.000373	0.996930	0.992735	0.992403	0.991795	$0.99485_{\pm 0.00370}$	5 / 5
3	Sqrt	0	Island	0.99545	0.99174	0.99470	0.99689	0.98887	$0.99353_{\pm 0.00321}$	5 / 5

K	Normalization	α_0	Penalty scope	Run 1	Run 2	Run 3	Run 4	Run 5	Mean ± s.d.	Feasible runs
26	None	0.5	Island	0.99764	0.99434	0.98047	0.99564	0.99124	$0.99187_{\pm 0.00678}$	5 / 5
26	Mean	0.5	Island	0.99750	0.97057	0.99400	0.99245	0.99722	$0.99035_{\pm 0.01126}$	5 / 5
26	Sqrt	0.5	Island	0.99694	0.99861	0.99859	0.99899	0.99925	$0.99848_{\pm 0.00090}$	5 / 5
26	Sqrt	0.5	Global	0.994779	0.998269	0.990736	0.998373	0.988948	$0.99422_{\pm 0.00430}$	5 / 5
26	Sqrt	0	Island	0.99522	0.94660	0.99756	0.99008	0.99722	$0.98534_{\pm 0.02186}$	5 / 5

H.2 Alpha Sensitivity on Tokamak Control

We vary the initial relaxation fraction while leaving the remaining CAP-Evolve configuration unchanged. Table H.2 reports the task-native, best-feasible mean relative D–T tracking error; lower is better. The $\alpha_0 = 0$ row corresponds to an update without alpha relaxation.

Table H.2: Alpha sensitivity.

α_0	Run 1	Run 2	Run 3	Run 4	Run 5	Mean ± s.d.	Feasible runs
0	0.07051	0.06155	0.06155	0.06684	0.09682	$0.07146_{\pm 0.01468}$	5 / 5
0.25	0.06451	0.06202	0.06540	0.07051	0.06341	$0.06517_{\pm 0.00324}$	5 / 5
0.5	0.06455	0.06155	0.07956	0.06353	0.06976	$0.06779_{\pm 0.00725}$	5 / 5
0.75	0.06424	0.10637	0.06157	0.06424	0.06209	$0.07170_{\pm 0.01942}$	5 / 5
1.0	0.06168	0.06155	0.07845	0.06697	0.06169	$0.06607_{\pm 0.00730}$	5 / 5

H.3 Fixed-Penalty Sweeps

We sweep scalar penalties on the common power-of-ten grid $\lambda = 10^n$ for the single-constraint versions of Circle Packing, Spectral Shape Optimization, Trajectory Optimization, and Tokamak Control. Each task block uses its task-native metric, with higher values preferred for Circle Packing and Spectral Shape Optimization and lower values preferred for Trajectory Optimization and Tokamak Control. For each run, we record both the best feasible retained program and the final program preferred by the fixed penalized fitness; penalized-output metrics are computed only over feasible native outputs. This separates the archive-based evaluation used throughout the paper from the native output of a fixed-penalty search.

Table H.3: Fixed-penalty sweep results.

Task	λ	Best-feasible metric: mean ± s.d.	Best-feasible runs	Penalized-output feasible runs	Penalized-output metric: mean ± s.d.	Selected for Table 1
Circle Packing (↑)	1	$0.98639_{\pm 0.01139}$	5 / 5	0 / 5	–	No
Circle Packing (↑)	10	$0.99598_{\pm 0.00354}$	5 / 5	5 / 5	$0.99598_{\pm 0.00354}$	Yes
Circle Packing (↑)	100	$0.98723_{\pm 0.02010}$	5 / 5	5 / 5	$0.98723_{\pm 0.02010}$	No
Spectral Shape Optimization (↑)	1	$1.01268_{\pm 0.01179}$	5 / 5	2 / 5	$1.00816_{\pm 0.01154}$	No
Spectral Shape Optimization (↑)	10	$1.02203_{\pm 0.01925}$	5 / 5	3 / 5	$1.02618_{\pm 0.02570}$	Yes
Spectral Shape Optimization (↑)	100	$1.01544_{\pm 0.01460}$	5 / 5	5 / 5	$1.01544_{\pm 0.01460}$	No
Trajectory Optimization (↓)	1	$1.36057_{\pm 0.00044}$	5 / 5	1 / 5	1.36077	No
Trajectory Optimization (↓)	10	$1.35924_{\pm 0.00260}$	5 / 5	5 / 5	$1.35924_{\pm 0.00260}$	Yes
Trajectory Optimization (↓)	100	$1.35924_{\pm 0.00342}$	5 / 5	5 / 5	$1.35924_{\pm 0.00342}$	No
Trajectory Optimization (↓)	1000	$1.35965_{\pm 0.00187}$	5 / 5	5 / 5	$1.35965_{\pm 0.00187}$	No
Trajectory Optimization (↓)	10000	$1.35953_{\pm 0.00278}$	5 / 5	5 / 5	$1.35953_{\pm 0.00278}$	No
Tokamak Control (↓)	1	$0.06714_{\pm 0.00879}$	5 / 5	4 / 5	$0.06328_{\pm 0.00193}$	No
Tokamak Control (↓)	10	$0.06526_{\pm 0.00175}$	5 / 5	5 / 5	$0.06526_{\pm 0.00175}$	Yes

For each task, Table 1 uses the coefficient with the best mean best-feasible task-native metric in this sweep. The selected scalar penalty is 10 for all four single-constraint tasks. The sweep is reported as a finite-grid diagnostic and does not establish a generally transferable fixed penalty.

I Search Trace Visualization

Figure I.1 visualizes a representative $K = 26$, active-count square-root-normalized Circle Packing run. The run begins from feasible packings and subsequently accumulates infeasible constructions with larger sums of radii. The upper panel traces the best feasible target ratio in each island and across the two islands, while the lower panel shows the corresponding penalty norms and alpha schedule; vertical markers identify a penalty update and the subsequent feasible improvement. Together, the panels show how feasible objective progress, island-specific penalty magnitudes, and relaxation evolve over the search.

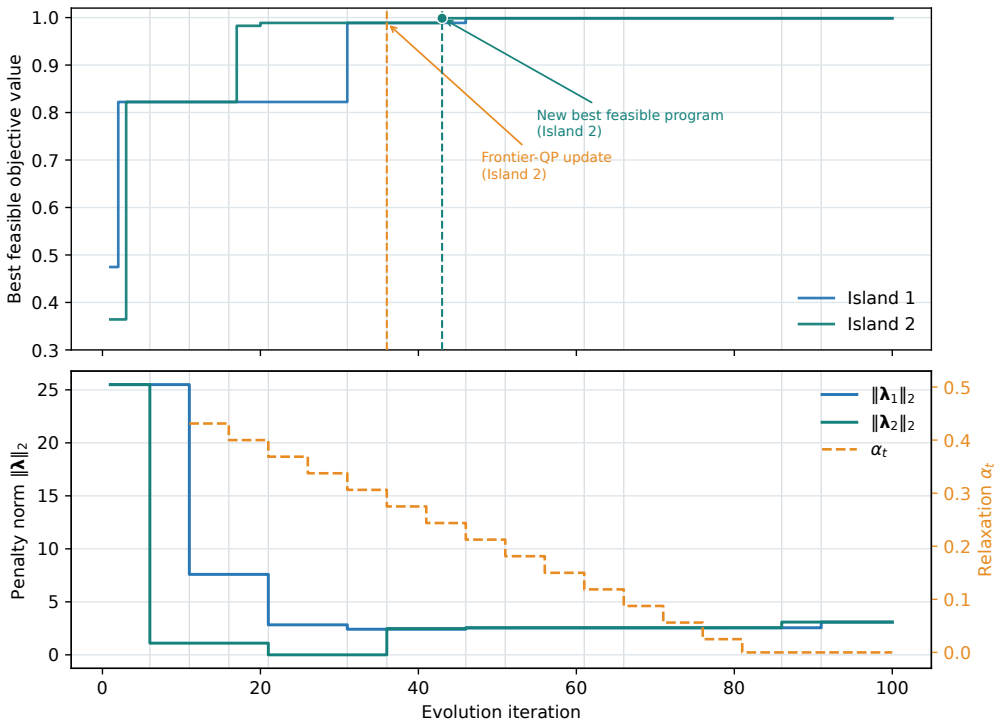


Figure I.1: Search trace for the representative $K = 26$ Circle Packing run. The upper panel reports island-level best feasible target ratios. The lower panel reports island-specific penalty norms and the alpha schedule. The orange marker denotes a Frontier-QP update in Island 2; the teal marker shows the subsequent new best feasible program from that island.