Rethinking LLM-Driven Heuristic Design: Generating Efficient and Specialized Solvers via Dynamics-Aware Optimization

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have advanced the field of Combinatorial Optimization through automated heuristic generation. Instead of relying on manual design, this LLM-Driven Heuristic Design (LHD) process leverages LLMs to iteratively generate and refine solvers to achieve high performance. However, existing LHD frameworks face two critical limitations: (1) Endpoint-only evaluation, which ranks solvers solely by final quality, ignoring the convergence process and runtime efficiency; (2) High adaptation costs, where distribution shifts necessitate re-adaptation to generate specialized solvers for new instance groups. To address these issues, we propose Dynamics-Aware Solver Heuristics (DASH), a framework that co-optimizes solver search mechanisms and runtime schedules guided by a convergence-aware metric, thereby identifying efficient and high-performance solvers. Furthermore, to mitigate expensive re-adaptation, DASH incorporates Profiled Library Retrieval (PLR). PLR efficiently archives specialized solvers concurrently with the evolutionary process, enabling cost-effective warm-starts for heterogeneous distributions. Experiments on four combinatorial optimization problems demonstrate that DASH improves runtime efficiency by over $3\times$, while surpassing the solution quality of state-of-the-art baselines across diverse problem scales. Furthermore, by enabling profile-based warm starts, DASH maintains superior accuracy under different distributions while cutting LLM adaptation costs by over 90%. Code is available at: https://anonymous.4open.science/r/Paper2026-DASH.

1 Introduction

Many fundamental problems in computing and engineering, including routing, scheduling, and chip placement, are combinatorial optimization problems (Korte and Vygen, 2008). In practice, high-performance solvers for these problems rely on carefully hand-crafted heuristics (Burke et al., 2013). Due to the extensive search space, executing such heuristic solvers is computationally expensive. Beyond the high designing cost, these heuristics often struggle with generalization: when the instance distribution changes (e.g., in size or density), sustaining performance typically requires re-design for re-adaptation (Wolpert and Macready, 2002).

LLM-Driven Heuristic Design (LHD) (Yao et al., 2025; Wu et al., 2025) alleviates this burden by automatically generating and improving solvers. Recent works have advanced this domain by establishing a fundamental workflow: an LLM-driven generate, evaluate, and select iteration, where LLMs refine solvers based on execution feedback (e.g., FunSearch (Romera-Paredes et al., 2024), EoH (Liu et al., 2024), and ReEvo (Ye et al., 2024)). However, evaluation requires repeatedly generating and executing time-consuming solvers, making LHD practical still faces two challenges.

Challenge 1: Generating efficient solvers. Most LHD frameworks (Zheng et al., 2025; Dat et al., 2025) select candidate solvers only by the final score (e.g., gap to an optimal or best-known value), thereby ignoring a key dimension: the convergence trajectory within the time. Crucially, early convergence often indicates the potential to further reduce the gap to the optimal (Hansen and Zilberstein, 1996), yet comparable final score makes it difficult to distinguish such efficient solvers. Some recent efficiency-aware LHD works have started to incorporate time objectives in solver selection (e.g., MEoH (Yao et al., 2025)). Nevertheless, relying solely on such efficiency signal fails to capture the convergence dynamics (e.g., early convergence, stable improvement, and long stagnation). As Figure 1 (left) shows, three solvers reach comparable final gaps with different trajectories, rendering them indistinguishable under endpoint-only evaluation.

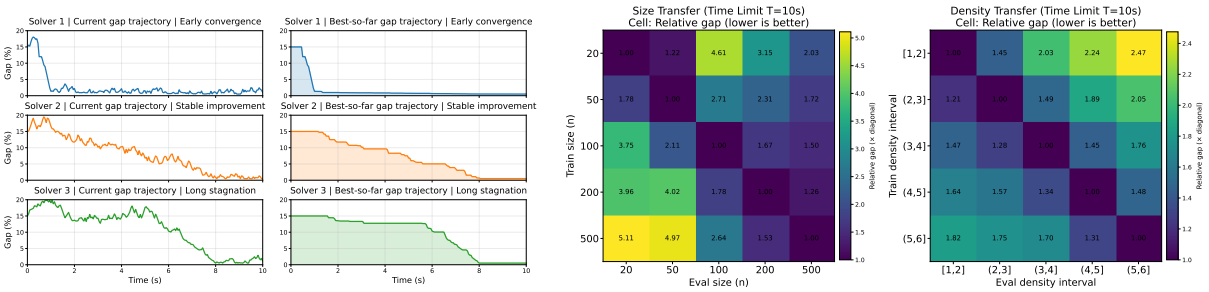

Figure 1: **Motivating Experiments on TSP tasks using Guided Local Search (Voudouris and Tsang, 1999) as solver backbone and under a 10s time limit.** (*Left*) Three solvers reach similar final gaps with different trajectories, rendering them indistinguishable under endpoint-only evaluation. Solver 1 converges earlier, achieving lower gaps at earlier times and leaving more of the time budget for subsequent improvements. (*Right*) Performance significantly degrades when solvers are transferred across shifts in problem size (node count) or density (node clustering), underscoring the necessity of generating specialized solvers for distinct instance distributions.

Challenge 2: Generating specialized solvers. LHD is costly as it requires repeatedly refining solvers on training instances (Guo et al., 2025). Some methods reduce this cost by reusing information from prior evaluations to filter redundant candidates, thereby cutting down solver runs (e.g., Hercules (Wu et al., 2025)). However, real-world instances are heterogeneous: a solver that works well on one instance group may degrade under distribution shifts in scale or other instance characteristics (Rice, 1976). Consequently, maintaining performance across diverse groups requires group-wise adaptation, leading to repeated LLM-driven iteration. Although prior work lowers cost per iteration, it remains inefficient under distribution shifts. As Figure 1 (right) illustrates, solvers optimized for one instance group can degrade significantly on another, resulting in costly re-adaptation.

These observations underscore the necessity of assessing a solver by the temporal evolution of its solution quality, framing execution as a dynamical process. Accordingly, we propose Dynamics-Aware Solver Heuristics (**DASH**), a framework that improves solver performance and efficiency by optimizing how the solver searches and how it spends runtime. Under this view, we decompose solver design into the search mechanism (e.g., update rules and guidance design) and the runtime schedule (e.g., time allocation across phases). To quantify these dynamics, we introduce the Trajectory-aware Lyapunov Decay Rate (**tLDR**) (Drazin, 1992), which measures the rate and consistency of convergence and guides the co-evolution of search mechanism and runtime schedule in DASH.

To mitigate re-adaptation costs, DASH further incorporates Profiled Library Retrieval (**PLR**). PLR decouples archiving from evolution: within a single evolutionary process, it archives group-specific solvers, while continuing to evolve the global solver based on average performance. This strategy constructs a diverse library without restarting the search, enabling cost-effective warm-starts. We validate DASH on four combinatorial optimization problems and demonstrate its generalizability across solver backbones. Results show that DASH improves runtime efficiency over 3× while surpassing state-of-the-art baselines in solution quality. By enabling profile-based warm starts, DASH maintains accuracy under different distributions while cutting LLM adaptation costs by over 90%.

Our contributions can be summarized as follows:

- We introduce the tLDR, a trajectory-aware metric that shifts evaluation from static endpoints to dynamic convergence efficiency, prioritizing fast and stable solvers.
- We propose DASH, a framework co-evolves search mechanism and runtime schedules. It incorporates Profiled Library Retrieval to efficiently harvest specialized solvers for cost-effective warm-starts.
- We validate DASH on four combinatorial optimization problems. Results show it reduces the optimality gap by over 50% while improving runtime efficiency by over 3× compared to state-of-the-art baselines.

2 Related Work

LLM-Driven Code Generation. Large Language Models (LLMs) have advanced code generation

from one-shot synthesis to iterative refinement. Recent approaches move beyond static text generation by incorporating execution feedback, utilizing test outcomes to verify correctness and guide code refinement (Wang et al., 2023; Zheng et al., 2024; Fakhoury et al., 2024). To handle complex dependencies, retrieval-augmented methods ground the generation in repository-level context (Zhang et al., 2023). Building on these, agent-based systems further orchestrate coding, retrieval, and testing loops to automate realistic software engineering tasks (Zhang et al., 2024; Jimenez et al., 2024).

LLM-Driven Heuristic Design. Combinatorial optimization problems typically rely on heuristic solvers (Garey and Johnson, 1979). Recently, LLMs have advanced LLM-Driven Heuristic Design (LHD) by establishing an iterative generate, evaluate, and select workflow to synthesize and refine solvers (Guo et al., 2025; Zheng et al., 2025; Dat et al., 2025). Notable frameworks include FunSearch (Romera-Paredes et al., 2024), which couples program mutation with evolutionary search based on execution feedback; EoH (Liu et al., 2024), which co-evolves natural language ideas and code implementations; and ReEvo (Ye et al., 2024), which employs reflective prompts to guide the search. However, most LHD methods select solvers solely based on endpoint metrics, ignoring the convergence process. Although some efficiency-aware variants incorporate runtime objectives, they typically reduce the dynamic trajectory to a few aggregated statistics (Yao et al., 2025). A complementary direction addresses evaluation costs by reusing mechanisms or predictor-based pruning to filter candidates (Wu et al., 2025; Guo et al., 2025). Although effective for reducing overhead, such pruning may limit search diversity and overlook promising solvers. DASH addresses these limitations by optimizing the full convergence trajectory for efficiency and employing profiled retrieval to enable cost-effective adaptation.

3 Method

3.1 Solver Runs as Time-Evolving Trajectory

We adopt a dynamical-systems view of solver execution under a time limit T. For an instance x, a solver $\pi = (\theta, \sigma)$ induces a time-evolving solution $z(\tau) \in \mathcal{Z}(x)$, where θ encodes the search mechanism (e.g., update rules and guidance design) and σ encodes the runtime schedule (e.g., time allocation across phases). In practice, solvers update their solutions in discrete steps, so we consider:

$$z_{k+1} = F(z_k; x, \theta, \sigma), \quad \tau_{k+1} = \tau_k + \Delta\tau_k, \quad (1)$$

where τ_k is the cumulative runtime after step k and $\Delta\tau_k$ is the measured time cost of that step.

3.2 Lyapunov Potential and Incumbent Trajectory

Lyapunov potential definition. To compare such trajectories across runs, we require a progress signal that measures the distance to a target solution $z^\star$ (e.g., an optimal or best-known solution) at each time. In dynamical-systems, Lyapunov functions formalize this idea as a generalized energy that is minimized at the target state (Drazin, 1992). Adopting this perspective, we define a proxy potential $V : \mathcal{Z}(x) \to \mathbb{R}_{\geq 0}$ satisfying:

$$V(z^\star) = 0, \qquad V(z) > 0 \text{ for } z \neq z^\star. \quad (2)$$

Incumbent trajectory. For a minimization objective $f_x(\cdot)$ with reference optimum $f^\star$ (exact or best-known), we measure the distance to optimality via the relative gap: $\mathrm{gap}_x(\tau) = \big(f_x(z(\tau)) - f_x^\star\big)/|f_x^\star|$. Consistent with Eq. (2), we set the time-dependent residual as the gap itself:

$$V(\tau) = \mathrm{gap}_x(\tau), \quad (3)$$

where $V(\tau)$ is short for $V(z(\tau))$. Since heuristic search trajectory is typically stochastic and non-monotone, we extract persistent progress via the incumbent (best-so-far) trajectory:

$$V_{\mathrm{best}}(\tau) = \min_{0 \leq u \leq \tau} V(u). \quad (4)$$

By construction, $V_{\mathrm{best}}(\tau)$ is non-increasing, providing a monotone progress signal that aligns with an energy-descent view of optimization dynamics.

We then project the incumbent trajectory into logarithmic space to obtain a scale-consistent notion of progress. In particular, many improvements in combinatorial optimization are naturally compared in relative terms (i.e., multiplicative reductions of the gap). Working in log space turns such multiplicative changes into additive decreases, making progress comparable across different residual magnitudes. To ensure well-defined values even when the optimum is reached (i.e., $V_{\mathrm{best}} = 0$), we employ a numerical lower bound $\delta > 0$:

$$\ell(\tau) = \ln \max\big(V_{\mathrm{best}}(\tau),\, \delta\big). \quad (5)$$

We use $\ell(T)$ as the terminal log-residual under the time limit T.

3.3 Trajectory-aware Lyapunov Decay Rate

Previous LHD methods based solely on the terminal gap at time T fail to capture optimization dynamics: two solvers may achieve similar $\ell(T)$ while exhibiting different early progress and different persistence in low-residual regimes. To summarize the incumbent evolution over the entire process, we compute the time-averaged log-residual:

$$J(T) = \frac{1}{T}\int_0^T \ell(\tau)\, d\tau. \tag{6}$$

A smaller $J(T)$ implies that the solver stays in low-residual states for a larger fraction, reflecting both early improvement and sustained convergence.

To convert this trajectory-aggregated quantity into an interpretable decay rate, we derive an effective slope. Specifically, we define an equivalent linear trajectory $\tilde{\ell}(\tau) = \ell(0) - k\tau$, anchored at the initial residual $\ell(0)$. We determine the decay slope k by requiring that this hypothetical linear path yields the exact same time-averaged performance $J(T)$ as the actual observed run over $[0, T]$.

The integral of this linear path is $T\ell(0) - \frac{1}{2}kT^2$. By equating this to observed integral $TJ(T)$ and solving for k, we define the **Trajectory-aware Lyapunov Decay Rate (tLDR)**:

$$\mathrm{tLDR}(T) = \frac{2}{T}\big(\ell(0) - J(T)\big). \tag{7}$$

$\mathrm{tLDR}(T)$ is the effective decay slope of incumbent log-residual trajectory over $[0, T]$ induced by the time-averaged log-residual $J(T)$. Equivalently, it is the constant slope of a linear surrogate $\tilde{\ell}(\tau) = \ell(0) - k\tau$ whose time-average matches the run. A larger tLDR indicates faster and more sustained reduction in log-residual throughout $[0, T]$, rather than progress concentrated near the end. For fair comparison, $\ell(0)$ is measured from the same initial state for each instance, reused across solver evaluations. (Discrete computation details and Lyapunov intuition are provided in Appendix C.)

3.4 Three Iteration Layers

The performance of a solver is governed by two interacting components: the mechanism θ and the schedule σ. However, directly optimizing $\pi = (\theta, \sigma)$ is challenging due to their complex coupling. To address this, we decompose optimization into three sequential layers. Across all layers, we employ a unified protocol based on the terminal log-residual $\ell(T)$, trajectory efficiency $\mathrm{tLDR}(T)$, and (for schedule shaping) solver runtime t_{run}. Decisions follow parameterized tolerance intervals that define acceptable trade-offs between solution quality and convergence speed. Threshold formulas are detailed in Appendix F, and prompt templates of different layers are provided in Appendix J.

3.4.1 Mechanism Discovery Layer (MDL)

MDL updates the mechanism θ while keeping the schedule σ fixed. Here, θ refers to the core search logic that determines how the solver changes its current solution state. In each iteration, the LLM produces a candidate mechanism θ' by editing the code of θ based on the parent and its evaluation feedback. We then evaluate θ' under a hierarchical selection criterion relative to its parent. A candidate is accepted if it achieves a lower terminal log-residual $\ell(T)$; when $\ell(T)$ is within a tolerance interval of the parent, we accept the one with higher trajectory efficiency $\mathrm{tLDR}(T)$.

3.4.2 Mechanism Consolidation Layer (MCL)

MCL controls the structural complexity of the evolving mechanism θ. Iterative evolution can lead to code bloat, where the mechanism accumulates brittle logic that overfits to specific instances. In this layer, the LLM refactors the mechanism by rewriting the code structure (e.g., merging duplicated branches and removing dead or redundant logic) while keeping its intended behavior. We employ a preservation criterion: a consolidated candidate θ' is accepted only if both $\ell(T)$ and $\mathrm{tLDR}(T)$ within the tolerance interval of the parent.

3.4.3 Schedule Shaping Layer (SSL)

SSL updates the schedule σ while keeping the mechanism θ fixed. The σ specifies how the solver runs over the time budget: which modules are invoked, in what order, and with what triggering rules and per-phase budgets. In each iteration, we provide the LLM with the parent schedule together with its evaluation traces (e.g., per-module runtime and the resulting convergence trajectory), and the LLM proposes a revised schedule σ' by adjusting module allocation and control parameters. We then evaluate σ' under the same protocol as its parent.

Two-stage shaping. SSL applies two consecutive stages: *Compression* to reduce wasted computational slack, followed by *Enhancement* to spend the recovered budget more effectively.

Stage 1: Compression. This stage targets computational slack in the execution chain. Given the

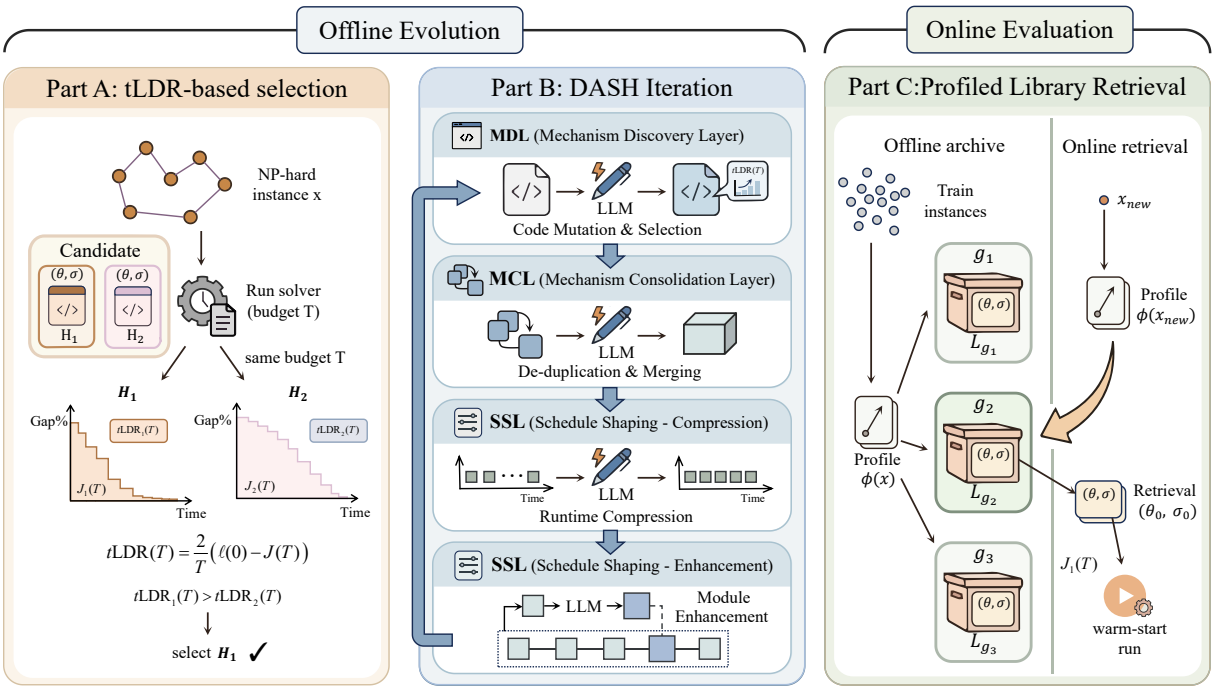

Figure 2: **Overview of the DASH framework.** Offline evolution co-evolves the solver across MDL, MCL, and SSL using terminal quality and trajectory efficiency for selection. In parallel, PLR maintains group-wise archives from evaluated candidates. At test time, PLR retrieves a group-specific solver to warm-start evalution.

parent trace, the LLM revises σ by shortening or removing low-efficiency phases and by retuning iteration limits or trigger conditions, so that the solver reaches comparable terminal quality with less solver runtime t_{run}. However, $\text{tLDR}(T)$ is typically compared under the same time budget T; once compression changes the actual runtime, trajectory comparisons based on $\text{tLDR}(T)$ become mismatched. Therefore, we use runtime t_{run} for schedule-level acceptance in this stage. A compressed schedule is accepted if it reduces t_{run} while keeping the terminal log-residual $\ell(t_{\text{run}})$ within the tolerance interval of the parent.

Stage 2: Enhancement. Once the schedule is compressed, this stage targets marginal gains in terminal quality by further modifying σ. Starting from the compressed schedule, the LLM revises σ by reallocating the compressed runtime budget across phases and adjusting schedule controls (e.g., module allocation and their triggering rules or iteration limits), so that more of the runtime is spent on phases that are most helpful for improving the incumbent (e.g., a more thorough perturbation or local-improvement setting) without increasing the runtime. An enhanced schedule is accepted only if it improves $\ell(T)$ and keeps $\text{tLDR}(T)$ within the tolerance interval of the parent.

4 DASH Framework

DASH co-evolves the solver mechanism θ and runtime schedule σ via iterative LLM-Driven edits across MDL, MCL, and SSL, and introduces Profiled Library Retrieval (PLR) to decouple archiving from population evolution within a single evolutionary process. We maintain a global population $\mathcal{P}$ of top-k solvers ranked by performance aggregated over all groups and a group-wise archive L_g that stores the top-k specialized solvers for each group g. During evolution, parents are sampled from $\mathcal{P}$. At test time, PLR selects a solver from $L_{g^\star}$ for profile-aware warm-starting. For fairness, each evolutionary run starts from the same task-specific initial solver. The workflow algorithm and implementation details are provided in Appendix E, and the main workflow is illustrated in Figure 2.

4.1 Instance Profiles and Instance Groups

To improve generalization across heterogeneous instances, we construct an offline training set $\mathcal{D}_{\text{train}}$ of randomly generated instances with diverse distributions. For each instance x, we compute a lightweight instance profile $\phi(x) \in \mathbb{R}^p$ summarizing key size and structural statistics, and partition $\mathcal{D}_{\text{train}}$ into G groups $\{\mathcal{D}_g\}_{g=1}^{G}$ based on profile similarity, each represented by a prototype ϕ_g. These

groups enable stratified batch sampling during evolution and provide the index used by profile-based retrieval. Feature definitions and instance generation protocols are provided in Appendix D.

4.2 Evaluation and Decoupled Archiving

We execute the iterative DASH loop on $\mathcal{D}_{\text{train}}$ to co-evolve the solver. To efficiently obtain both global and group-specialized solvers, we employ a decoupled archiving strategy. In each iteration, we evaluate candidates on a sampled batch $\mathcal{B}$, composed of batches $\mathcal{B}_g$ sampled from each group $\mathcal{D}_g$:

$$\mathcal{B} = \cup_{g=1}^{G} \mathcal{B}_g, \quad \mathcal{B}_g = \text{Sample}(\mathcal{D}_g, m). \quad (8)$$

Based on these evaluations, we (i) update the size-k global population using layer-wise acceptance on batch means, replacing the lowest-ranked solver if accepted, and (ii) update each group archive L_g (top-k) using the candidate's batch-mean performance on $\mathcal{B}_g$. Archive updates are independent of population acceptance: every evaluated candidate can be inserted into L_g if it outperforms the lowest-ranked entry, keeping only the top-k solvers.

4.3 Online Profiled Library Retrieval (PLR)

Given a query instance x_{new}, we compute and normalize its instance profile $\phi(x_{\text{new}})$ using the training set statistics. We then identify the most similar group $g^\star$ by minimizing the Euclidean distance to the prototype profiles:

$$g^\star = \arg \min_{g \in \{1,\dots,G\}} \left\| \phi(x_{\text{new}}) - \phi_g \right\|_2, \quad (9)$$

where ϕ_g is the prototype profile of group g. Once the best-matching group is identified, we retrieve best solver $(\theta^\star, \sigma^\star)$ from the corresponding group archive $L_{g^\star}$. This warm-starts the new instance with a solver specialized to similar profile statistics.

5 Experiment

Dataset. For synthetic Euclidean TSP, we generate instances with $n \in \{20, 50, 100, 500, 1000\}$ nodes by sampling node coordinates independently from $[0, 1]$ (Liu et al., 2024). We also evaluate on standard benchmark, including TSPLIB (TSP) (Reinelt, 1991), CVRPLIB (CVRP) (Uchoa et al., 2017), and OR-Library (MKP) (Beasley, 1990). For online BPP, we generate instances following the Weibull sampling protocol used in MEoH (Yao et al., 2025). Detailed task definitions and instance selection protocols are provided in Appendix A.

Solver Backbone. For TSP, we use Guided Local Search (GLS) (Voudouris and Tsang, 1999) with 10 s per instance (60 s for TSP1000) and at most 1000 internal iterations. We also instantiate DASH on Iterated Local Search (ILS) (Glover and Kochenberger, 2003) and a Python implementation of LKH (Helsgaun, 2000). For CVRP/MKP we use Ant Colony Optimization (ACO) (Dorigo et al., 2007), and for online BPP we use Greedy Online Assignment (GOA) (Yao et al., 2025).

Baseline Methods. We compare (i) dedicated solvers (Concorde (Applegate et al., 2011), LKH3 (Helsgaun)), (ii) handcrafted heuristics (LS (Aarts and Lenstra, 2018), GLS (Voudouris and Tsang, 1999), KGLS (Arnold and Sörensen, 2019)), and (iii) LHD frameworks (FunSearch (Romera-Paredes et al., 2024), ReEvo (Ye et al., 2024), EoH (Liu et al., 2024), MEoH (Yao et al., 2025), Hercules (Wu et al., 2025)). All LHD baselines use GPT-5-mini (OpenAI, 2025). Detailed training and evaluation settings are in Appendix A.

Evaluation Metrics. We report *Gap* and *Time*, where $\text{Gap}(\%) = \frac{f - f^\star}{|f^\star|} \times 100$ and *Time* is the runtime per instance under the specified budget.

Our experiments are designed as follows:

- **RQ1 (Main Results):** How does DASH compare to prior LHD frameworks in solution quality and runtime efficiency?
- **RQ2 (Ablation):** How does each component of DASH contribute to runtime efficiency and solution quality?
- **RQ3 (Generalization):** How well does DASH perform across solver backbones, distribution shifts, and base LLMs?
- **RQ4 (Further Study):** How does DASH compare to dedicated solvers and LHD baselines under time matched and unconstrained settings (Detailed in Appendix H)?

5.1 Main Results (RQ1)

We evaluate DASH against state-of-the-art baselines on TSP (Table 1). Compared to the GLS backbone, DASH reduces the TSP1000 gap from 17.713% to 1.004% (over 90%), validating the effectiveness of co-evolving mechanisms and schedules. Crucially, the comparison highlights the advantage of trajectory-aware optimization. On larger scales (e.g., TSP1000), endpoint-driven baselines (ReEvo, EoH) typically consume the full 60 s budget even when marginal gains diminish. In contrast, DASH reduces computational slack via SSL

Method	TSP 20		TSP 50		TSP 100		TSP 500		TSP 1000	
	Gap (%)	Time (s)	Gap (%)	Time (s)	Gap (%)	Time (s)	Gap (%)	Time (s)	Gap (%)	Time (s)
Concorde	0.000	0.011	0.000	0.055	0.000	0.235	0.000	7.315	0.000	32.083
LKH3	0.000	0.021	0.000	0.071	0.012	0.122	0.022	3.038	0.035	12.154
LS	4.267	0.001	4.148	0.009	5.365	0.299	8.161	7.359	13.429	60.000
GLS	0.002	0.753	0.039	3.088	2.084	9.861	9.677	10.114	17.713	60.000
KGLS	0.007	1.082	0.033	3.604	1.443	9.370	5.541	10.002	11.734	60.007
FunSearch	**0.000**	0.909	**0.000**	4.001	0.225	10.212	2.080	10.095	4.233	60.513
ReEvo	**0.000**	0.763	**0.000**	3.160	0.191	10.095	2.040	10.167	3.490	60.737
EoH	**0.000**	1.031	**0.000**	4.800	0.122	10.847	2.387	10.086	4.558	59.263
MEoH	**0.000**	0.619	0.620	2.928	0.524	3.280	3.383	6.734	4.931	40.556
Hercules	**0.000**	0.823	**0.000**	3.542	0.169	10.970	1.935	10.684	3.312	58.656
DASH (Ours)	**0.000**	**0.094**	0.029	**0.112**	**0.086**	**1.136**	**0.974**	**3.680**	**1.004**	**12.501**

Table 1: **Performance comparison on TSP.** Gap (%) and Time (s) are averaged over 100 instances per scale. Each run limits 10s per instance (except TSP1000: 60s).

Method	CVRP 100		CVRP 500		BPP C100		BPP C500		MKP 100		MKP 500	
	Gap(%)	Time(s)	Gap(%)	Time(s)	Gap(%)	Time(s)	Gap(%)	Time(s)	Gap(%)	Time(s)	Gap(%)	Time(s)
FunSearch	0.625	4.253	5.857	10.152	1.854	0.851	1.459	4.200	2.155	1.853	2.950	4.506
EoH	0.425	5.104	**3.120**	10.508	1.152	1.055	1.687	4.103	1.450	2.106	1.854	4.959
ReEvo	0.368	5.821	4.658	10.903	**0.450**	1.254	1.157	4.502	1.105	2.400	1.328	5.206
MEoH	0.408	3.500	5.784	6.807	1.722	0.956	1.253	3.105	1.680	1.659	1.554	3.851
Hercules	0.375	5.455	4.582	9.658	0.623	1.180	1.087	4.354	1.081	2.306	1.280	5.059
DASH (Ours)	**0.334**	**2.013**	3.935	**3.442**	0.463	**0.433**	**0.835**	**1.486**	**0.875**	**0.597**	**0.998**	**0.832**

Table 2: **Performance comparison on CVRP/BPP/MKP.** Gap (%) and Time (s) are averaged over 100 instances per scale. Each run limits 10s per instance.

Variant	TSP 100		CVRP 100	
	Gap (%)	Time (s)	Gap (%)	Time (s)
DASH (full)	**0.086**	**1.136**	**0.334**	**2.013**
w/o tLDR	0.452	2.200	0.385	4.859
w/o SSL-1	0.105	8.505	0.362	6.201
w/o SSL-2	0.165	1.002	0.392	1.920
w/o PLR	0.155	1.980	0.450	2.963

Table 3: **Ablation study on DASH components.** We report Gap (%) and Time (s) on TSP100 and CVRP100.

(compression stage), achieving better gaps while using 12.5 s on average. Meanwhile, the tLDR-based criterion preserves convergence behavior during schedule shaping, so efficiency gains do not compromise terminal quality. While MEoH reduces runtime, it yields noticeably worse gaps on TSP1000 (4.931%). DASH avoids this trade-off by optimizing the full trajectory.

Table 2 shows consistent improvements on CVRP/BPP/MKP. DASH attains the lowest runtime across all six settings while achieving the best or competitive gaps. On CVRP-500, compared to ReEvo and Hercules, DASH reduces runtime from 10 s to 3.44 s while also improving the gap, suggesting that schedule shaping mitigates low-yield computation across diverse problem structures.

5.2 Ablation Study (RQ2)

Table 3 validates the contribution of DASH's components. Removing SSL-1 causes the largest slowdown on both tasks, indicating that most runtime reduction comes from schedule compression. Without tLDR in selection degrades both runtime and solution quality, consistent with its role in favoring solvers with efficient early convergence. In contrast, removing SSL-2 mainly degrades the final gap with only minor runtime changes, matching its role as a quality-oriented enhancement stage. Finally, disabling PLR worsens both gap and time, showing that profile-based retrieval improves group-specific solver selection at inference. Additional analyses of the iteration layers and the impact of replacing tLDR with alternative trajectory metrics are reported in Appendix B.

Figure 3 visualizes the evolutionary dynamics on TSP100 across repeated runs. Endpoint-only baselines (FunSearch, ReEvo, EoH) typically remain near the fixed budget, whereas DASH reduces both the best-so-far gap and the runtime of the selected solver over evaluations. MEoH also reduces runtime, but its gap curve flattens earlier, suggesting weaker refinement. In contrast, DASH achieves

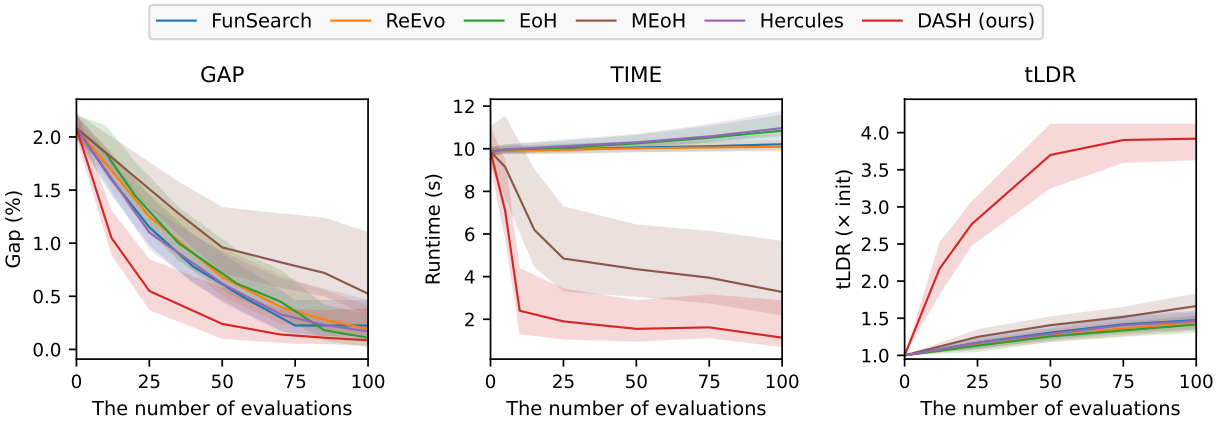

Figure 3: **Evolutionary dynamics on TSP100.** Across 5 independent runs (100 evaluations each), we track the best-so-far gap (left) and the corresponding runtime (middle) and tLDR (right). Lines show the mean and shaded bands show the variability across runs. Higher tLDR indicates more efficient anytime improvement.

Method	TSPLIB-small		TSPLIB-large	
	Gap(%)	Time(s)	Gap(%)	Time(s)
GLS	1.898	0.842	11.819	20.340
ILS	1.518	0.884	9.455	21.357
LKH	1.329	1.010	8.273	24.408
DASH+GLS	0.122	0.263	1.024	4.520
DASH+ILS	0.102	0.276	0.857	4.746
DASH+LKH	0.092	0.315	0.770	5.424

Table 4: **Transferability of DASH across solver frameworks on TSPLIB.** We report Gap (%) and Time (s) on small (14–200) and large (201–1000) instances.

Method	**Train**		**Test**		**Tokens (k) In/Out**
	Gap	Time	Gap	Time	
MEoH	1.499	9.623	3.558	13.961	44.7/27.3
MEoH (+Iters)	1.201	8.902	2.980	14.056	229.5/131.0
MEoH (+Groups)	0.620	4.887	1.550	10.967	472.9/280.6
Hercules	0.988	17.901	2.507	20.609	143.4/31.2
Hercules (+Iters)	0.900	18.420	1.950	23.859	711.4/155.7
Hercules (+Groups)	0.510	17.381	0.480	24.002	1401.1/322.3
DASH	**0.419**	**1.905**	**0.442**	**1.775**	**115.0/58.8**

Table 5: **Generalization Cost on TSPLIB (14–1000) with 100 evaluations.** (+Iters) increases the evaluation to 5× (500 evaluations); (+Groups) applies the same instance grouping protocol as DASH.

lower runtime while improving the best-so-far gap, consistent with the steadily increasing tLDR (right), which favors efficient anytime improvement rather than endpoint-only optimization.

5.3 Generalizability and Robustness (RQ3)

Table 4 shows that DASH consistently improves all three TSP backbones (GLS/ILS/LKH) on TSPLIB, reducing both gap and runtime on small and large instances. The gains across distinct backbones indicate that DASH acts as a general optimizer rather than being tied to a specific solver.

Table 5 studies generalization under a distribution shift: all methods are trained under the same setting as Table 1 (100 evaluations) and tested on TSPLIB. For baselines, (+Iters) increases the evaluation budget to 5 ×, and (+Groups) trains group-specific solvers using the same grouping protocol as DASH. Base LHD baselines degrade on TSPLIB, and mitigating the shift via (+Iters) or (+Groups) is token-expensive; (+Groups) can also increase test-time runtime when the selected specialists are slower. In contrast, DASH achieves similar train and test gaps with much lower test-time runtime, without a substantial increase in token usage required by (+Iters) or (+Groups), indicating that PLR enables group-aware specialization efficiently. Additional results on instantiating DASH with different base LLMs are reported in Appendix B.

6 Conclusion

We revisit LLM-Driven Heuristic Design from a dynamics perspective, evaluating solvers by their convergence trajectories rather than only terminal gap. We introduce the Trajectory-aware Lyapunov Decay Rate (tLDR), a metric computed from execution traces that captures the rate and consistency of convergence throughout the run. Guided by tLDR, DASH co-evolves search mechanisms and runtime schedules through MDL/MCL/SSL under a unified acceptance protocol. To reduce re-adaptation under heterogeneous instance groups, DASH incorporates Profiled Library Retrieval (PLR), which decouples group-wise archiving from population evolution to enable profile-aware warm starts. Across four combinatorial optimization problems, DASH improves solution quality and runtime efficiency, and transfers across multiple solver backbones.

7 Limitations

While DASH improves solver efficiency and reduces re-adaptation cost, it still has several practical limitations. First, DASH remains execution-driven: each candidate must be run to obtain convergence traces for tLDR and schedule feedback. This preserves selection reliability, but it also makes the overall training cost scale with solver runtime and the evaluation budget. Reducing this cost through more trace reuse, lightweight surrogates, or partial evaluations is a promising direction. Second, DASH depends on the underlying base LLM for proposing code edits. Although we observe stable behavior across multiple base models in our sensitivity study, stronger coding models and better prompt control may further improve reliability and reduce variance. Third, the potential of DASH can be further explored. In our comparison to dedicated solvers, DASH achieves strong gap–time trade-offs, but dedicated solvers are often the result of years of task-specific engineering and tuning. A natural next step is to characterize when DASH can further optimize a backbone to match or even surpass such specialized solvers, for example under specific instances, runtime budgets, or deployment constraints where trajectory-aware schedule shaping provides larger gains.

References

Emile Aarts and Jan Karel Lenstra. 2018. *Local Search in Combinatorial Optimization*. Princeton University Press.

David L Applegate, Robert E Bixby, Vašek Chvátal, and William J Cook. 2011. *The Traveling Salesman Problem: A Computational Study*. Princeton university press.

Florian Arnold and Kenneth Sörensen. 2019. Knowledge-guided local search for the vehicle routing problem. *Computers & Operations Research*, 105:32–46.

John E Beasley. 1990. Or-library: distributing test problems by electronic mail. *Journal of the operational research society*, 41(11):1069–1072.

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.

Pham Vu Tuan Dat, Long Doan, and Huynh Thi Thanh Binh. 2025. Hsevo: Elevating automatic heuristic design with diversity-driven harmony search and genetic algorithm using LLMs. In *Thirty-Ninth AAAI Conference on Artificial Intelligence, AAAI 2025*, pages 26931–26938.

DeepSeek-AI, Aixin Liu, Aoxue Mei, and 1 others. 2025. Deepseek-v3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Marco Dorigo, Mauro Birattari, and Thomas Stutzle. 2007. *Ant Colony Optimization*, volume 1. IEEE.

Philip G Drazin. 1992. *Nonlinear systems*. 10. Cambridge University Press.

Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. LLM-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Transactions on Software Engineering*, 50(9):2254–2268.

Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

Fred W Glover and Gary A Kochenberger. 2003. *Iterated Local Search*, volume 57.

Google DeepMind. 2025. Gemini 3 pro model card. Technical report, Google DeepMind.

Shuhan Guo, Nan Yin, James Kwok, and Quanming Yao. 2025. Nested-refinement metamorphosis: Reflective evolution for efficient optimization of networking problems. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 17398–17429.

Eric A. Hansen and Shlomo Zilberstein. 1996. Monitoring the progress of anytime problem-solving. In *Thirteenth National Conference on Artificial Intelligence, AAAI 1996*, pages 1229–1234.

Keld Helsgaun. LKH-3: Lin–kernighan–helsgaun TSP solver (version 3).

Keld Helsgaun. 2000. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *Twelfth International Conference on Learning Representations, ICLR 2024*.

Bernhard Korte and Jens Vygen. 2008. *Combinatorial optimization: theory and algorithms*. Springer.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. 2024. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Forty-first International Conference on Machine Learning, ICML 2024*.

Meta. 2024. Meta llama 3.3 model card.

OpenAI. 2025. Gpt-5 system card. Technical report, OpenAI.

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. In *Advances in Computers*, volume 15, pages 65–118. Elsevier.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, and 1 others. 2024. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475.

Eduardo Uchoa, Diego Pecin, Artur Pessoa, Marcus Poggi, Thibaut Vidal, and Anand Subramanian. 2017. New benchmark instances for the capacitated vehicle routing problem. *European Journal of Operational Research*, 257(3):845–858.

Chris Voudouris and Edward Tsang. 1999. Guided local search and its application to the traveling salesman problem. *European Journal of Operational Research*, 113(2):469–499.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2023. Execution-based evaluation for open-domain code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*.

David H Wolpert and William G Macready. 2002. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82.

Xuan Wu, Di Wang, Chunguo Wu, Lijie Wen, Chunyan Miao, Yubin Xiao, and You Zhou. 2025. Efficient heuristics generation for solving combinatorial optimization problems using large language models. In *The 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2025*.

An Yang, Anfeng Li, Binyuan Hui, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Shunyu Yao, Fei Liu, Xi Lin, Zhichao Lu, Zhenkun Wang, and Qingfu Zhang. 2025. Multi-objective evolution of heuristic using large language model. In *Thirty-Ninth AAAI Conference on Artificial Intelligence, AAAI 2025*.

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 NeurIPS 2024*.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Conference on Empirical Methods in Natural Language Processing, EMNLP 2023*.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *62nd Annual Meeting of the Association for Computational Linguistics, ACL 2024*.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. OpenCodeInterpreter: Integrating code generation with execution and refinement. In *Findings of the Association for Computational Linguistics: ACL 2024*.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. 2025. Monte carlo tree search for comprehensive exploration in LLM-based automatic heuristic design. In *Forty-second International Conference on Machine Learning, ICML 2025*.

A Experimental Setting

Setting of DASH. We use GPT-5-mini (OpenAI, 2025) with temperature set to 0.7 as the base LLM for DASH. For offline evolution, we perform 100 solver evaluations in total. We set the population size and the per-group archive size to $k = 5$. We partition the evolution set into $G = 10$ profile groups; in each evaluation, we sample $m = 3$ instances from each group, forming a union mini-batch $\mathcal{B}$ with $|\mathcal{B}| = Gm = 30$.

Runtime accounting. All methods are evaluated under the same time budget T, whereas DASH and MEoH may terminate earlier after time compression; we always report the measured wall-clock runtime of each methods.

Training and test set construction. For each task, we generate an instance pool and split it into a training set $\mathcal{D}_{\text{train}}$ and a test set $\mathcal{D}_{\text{test}}$. $\mathcal{D}_{\text{train}}$ is used only for DASH evolution: (i) computing instance profiles and K-means grouping into G=10 groups, and (ii) sampling per-iteration mini-batches (with m=3 per group, $|\mathcal{B}|$=30) for solver evaluation and acceptance. $\mathcal{D}_{\text{test}}$ is never used for selection, archiving, or PLR updates; it is used only for reporting final results. When a public benchmark is available, $\mathcal{D}_{\text{test}}$ includes both an in-distribution synthetic split ($\mathcal{D}_{\text{test-ID}}$) and an out-of-distribution benchmark split ($\mathcal{D}_{\text{test-OOD}}$).

TSP (GLS/ILS/LKH backbones). $\mathcal{D}_{\text{train}}$ contains 2,000 synthetic Euclidean instances: $n \in$

$\{20, 50, 100, 500, 1000\}$ with 400 instances per scale. Within each scale, coordinates are sampled from a mixture of five patterns (uniform, clustered mixtures, jittered grids, ring, elongated rectangles) with equal probability. $\mathcal{D}_{\text{test-ID}}$ contains 500 synthetic instances (100 per scale) generated with independent seeds. $\mathcal{D}_{\text{test-OOD}}$ contains 77 TSPLIB instances with $n \leq 1000$ to assess transfer beyond synthetic distributions.

CVRP (ACO backbones). $\mathcal{D}_{\text{train}}$ contains 1,000 synthetic Euclidean instances: $n \in \{100, 500\}$ with 500 instances per scale. Customer/depot coordinates are sampled uniformly from $[0, 1]^2$; demands are sampled i.i.d. from $\{1, \ldots, 9\}$; vehicle capacity is set to $Q = \lceil 0.5n \rceil$. $\mathcal{D}_{\text{test-ID}}$ contains 200 synthetic instances (100 per regime) with independent seeds. $\mathcal{D}_{\text{test-OOD}}$ is drawn from CVRPLIB: 40 instances for CVRP100 ($n \in [100, 200]$) and 20 instances for CVRP500 ($n \in [500, 600]$).

BPP (GOA backbones). We follow the standard Weibull sampling protocol used in MEoH for online BPP. $\mathcal{D}_{\text{train}}$ contains 1,000 Weibull instances with 5,000 items: 500 instances with capacity C=100 and 500 with C=500 (independent seeds). $\mathcal{D}_{\text{test}}$ contains 200 instances (100 per capacity) with independent seeds.

MKP (ACO backbones). $\mathcal{D}_{\text{train}}$ contains 1,000 synthetic MKP instances with m=5 constraints: 500 instances of (n, m)=$(100, 5)$ and 500 instances of $(500, 5)$. Profits and weights are sampled i.i.d. from $\{1, \ldots, 100\}$; each capacity is set to 50% of the sum of weights in the corresponding constraint. $\mathcal{D}_{\text{test}}$ uses the OR-Library benchmarks: all 30 instances from `mknapcb1.txt` ($(100, 5)$) and all 30 instances from `mknapcb3.txt` ($(500, 5)$).

PLR inference. At test time, we compute the profile $\phi(x)$ for a new instance and retrieve its nearest group prototype; we then deploy the best archived group-specific solver without additional LLM calls.

B Ablation Study

B.1 Ablation Study on Iteration Layers

We further ablate the three iteration layers in DASH. Table 6 reports the effect of removing each layer on TSP100 and CVRP100. Removing MDL leads to the largest degradation in gap, indicating that mechanism discovery is the main source of solution quality improvements. In contrast, removing MCL mainly increases runtime while only moderately affecting gap, consistent with MCL removing redundant branches and improving efficiency without changing the schedule. Finally, removing SSL (both compression and enhancement) increases runtime to near the fixed budget, showing that schedule shaping is the primary driver of runtime reduction.

Variant	TSP 100		CVRP 100	
	Gap (%)	Time (s)	Gap (%)	Time (s)
DASH (full)	**0.086**	**1.136**	**0.334**	**2.013**
w/o MDL	0.625	0.950	0.812	1.845
w/o MCL	0.114	1.680	0.358	2.550
w/o SSL (Full)	0.158	9.105	0.380	8.950

Table 6: **Ablation study on DASH Iteration Layers.** We report Gap (%) and Time (s) on TSP100 and CVRP100. w/o SSL (Full) indicates removing both compression and enhancement stages, using a fixed schedule throughout.

B.2 Trajectory Metrics Selection

To justify the design of the tLDR, we study how the choice of trajectory metric affects the evolutionary selection. Concretely, we replace tLDR with an alternative metric κ, while keeping the rest of DASH unchanged. We compare tLDR against three alternatives:

- **Terminal Time:** uses runtime t_{run} as the efficiency signal.
- **Time-to-10%:** the time to reach $0.1 \cdot \text{Gap}(0)$ on the incumbent trajectory; if threshold is not reached within the budget, we set it to T.
- **Linear AUC:** the normalized area under the incumbent gap curve in linear space, $\frac{1}{T}\int_0^T \text{Gap}_{\text{best}}(\tau)\, d\tau$ (smaller is better).

Metric Variant	TSP 100		CVRP 100	
	Gap (%)	Time (s)	Gap (%)	Time (s)
DASH (tLDR)	**0.086**	1.136	**0.334**	2.013
Terminal Time	0.585	**0.850**	0.852	**1.420**
Time-to-10%	0.242	1.020	0.425	1.880
Linear AUC	0.130	1.105	0.468	1.950

Table 7: **Trajectory metrics for selection.** DASH variants obtained by replacing tLDR with alternative trajectory metrics in the selection rules.

Replacing tLDR with alternative metrics changes the gap–time trade-off. Using *Terminal Time* yields the lowest runtime but substantially worse gaps, indicating that prioritizing runtime alone tends to select solvers with weaker refinement. *Time-to-10%* and *Linear AUC* provide intermediate trade-offs, improving over *Terminal Time* in gap but still inferior to tLDR. Overall, tLDR

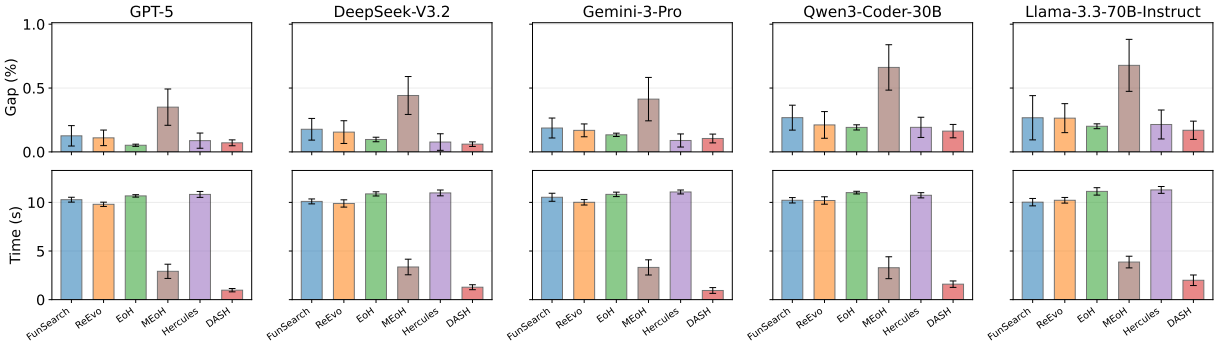

Figure 4: **Base-model sensitivity on TSP100.** Using five base LLMs for solver generation (5 runs, 100 evaluations each), we report the final best-so-far gap (top) and the runtime of that best solver (bottom).

achieves the best gaps with only moderate increases in runtime, suggesting that log-space trajectory evaluation provides a better-balanced selection signal for fast progress and continued improvement.

B.3 Sensitivity to Base LLMs

Figure 4 reports base-model sensitivity on TSP100. We instantiate solver generation with five base LLMs, including GPT-5 (OpenAI, 2025), DeepSeek-V3.2 (DeepSeek-AI et al., 2025), Gemini-3-Pro (Google DeepMind, 2025), Qwen3-Coder-30B (Yang et al., 2025), and Llama-3.3-70B-Instruct (Meta, 2024) (5 runs, 100 evaluations each). We report the final best-so-far gap and the runtime of the corresponding best solver. Across models, DASH maintains consistently low gaps and short runtimes with modest variance, indicating low sensitivity to the choice of base LLM. This is consistent with the trajectory-based selection in DASH remaining effective under different code-generation models.

C Lyapunov Intuition and Discrete Computation of tLDR

C.1 Lyapunov intuition

We view solver execution under a time budget as a dynamical process that induces a solution trajectory. To compare trajectories across solver configurations, we require (i) a non-negative residual that tracks progress toward a reference target, evaluated in log space to make progress comparable across residual magnitudes, and (ii) a single, rate-oriented summary of how quickly this residual decreases over time.

Lyapunov potential. We use a non-negative progress signal that measures the residual to a target solution $z^\star$ (optimal or best-known). Inspired by Lyapunov functions in dynamical systems, which are minimized at the target state, we consider a Lyapunov-inspired proxy potential: $V : \mathcal{Z}(x) \to \mathbb{R}_{\geq 0}$ satisfying

$$V(z^\star) = 0, \qquad V(z) > 0 \text{ for } z \neq z^\star. \tag{10}$$

Idealized exponential decay. Let $\Phi(\tau)$ denote such a non-negative residual. A common interpretation of convergence speed is motivated by an idealized exponential decrease:

$$\Phi(\tau) \approx \Phi(0)\, e^{-\kappa\tau}, \quad \kappa > 0, \tag{11}$$

which becomes approximately linear in log space by $\varphi(\tau) \triangleq \ln \Phi(\tau)$:

$$\varphi(\tau) \approx \varphi(0) - \kappa\tau. \tag{12}$$

Under this idealization, the slope in log space corresponds to an effective decay rate.

Equivalent constant log-slope (tLDR). To obtain a single rate from an arbitrary log-residual trace $\ell(\tau)$, we map it to an equivalent linear surrogate $\tilde{\ell}(\tau) = \ell(0) - k\tau$ whose time average matches $J(T)$:

$$J(T) := \frac{1}{T}\int_0^T \tilde{\ell}(\tau)\, d\tau = \ell(0) - \frac{kT}{2}.$$

Solving for k yields:

$$k = \frac{2(\ell(0) - J(T))}{T},$$

which is reported as tLDR (Eq. (7)).

Incumbent trajectory and log-residual. Heuristic search trajectories can be stochastic and non-monotone, so the instantaneous residual may fluctuate. To remove backtracking and obtain a monotone non-increasing trace, we use the incumbent

(best-so-far) trajectory:

$$V_{\text{best}}(\tau) = \min_{0 \le u \le \tau} V(u). \tag{13}$$

We then map it to log space (Eq. (5) in the main text), using a small floor δ to ensure the transform is well-defined, so that multiplicative improvements in the residual become additive decreases in $\ell(\tau)$. The resulting $\ell(\tau)$ is summarized over $[0, T]$ by $J(T)$ and mapped to an equivalent constant log-slope (tLDR) via Eq. (7).

C.2 Discrete computation of tLDR from logged traces

In our implementation, the incumbent trajectory $V_{\text{best}}(\tau)$ is tracked as a piecewise-constant function. Let $0 = \tau_0 < \tau_1 < \cdots < \tau_m = T$ be the recorded timestamps (e.g., corresponding to incumbent-update events or periodic logging). If the solver terminates early at $\tau_{\text{end}} < T$, we set the last timestamp to $\tau_m = T$ and extend the incumbent value as constant on $[\tau_{\text{end}}, T]$, i.e., $V_{\text{best}}(\tau) = V_{\text{best}}(\tau_{\text{end}})$ for $\tau \in [\tau_{\text{end}}, T]$. Here T denotes the runtime horizon used for evaluating the trace; in any parent–candidate comparison, both runs are evaluated under the same T.

Consistent with the log-residual definition in Eq. (5), we compute the discrete log-values as

$$\ell_i = \ln \max\big(V_{\text{best}}(\tau_i),\, \delta\big), \tag{14}$$

where δ is the numerical stability constant used in the main text.

Exact Riemann-sum form for $J(T)$. Since $V_{\text{best}}(\tau)$ (hence $\ell(\tau)$) is piecewise constant between recorded timestamps, the time-averaged log-residual admits the exact Riemann-sum form over intervals $[\tau_i, \tau_{i+1})$:

$$\widehat{J}(T) = \frac{1}{T} \sum_{i=0}^{m-1} \ell_i \,(\tau_{i+1} - \tau_i). \tag{15}$$

Discrete tLDR estimator. Substituting $\widehat{J}(T)$ into the linear-equivalent slope formula (Eq. (7)) yields the discrete tLDR estimator:

$$\widehat{\text{tLDR}}(T) = \frac{2}{T}\big(\ell_0 - \widehat{J}(T)\big). \tag{16}$$

This estimator is insensitive to the solver's internal iteration rate or logging frequency, provided that (i) all incumbent changes are captured and (ii) the time deltas $(\tau_{i+1} - \tau_i)$ reflect wall-clock durations.

D Instance Profiling and Grouping Details

D.1 Instance Profiling Features

For each instance $x \in \mathcal{D}_{\text{train}}$, we compute a lightweight profile vector $\phi(x) \in \mathbb{R}^p$ that summarizes structural statistics.

TSP/CVRP (Euclidean routing). We include: (i) scale features (node count n; capacity-related scalars for CVRP), (ii) distance statistics (e.g., mean/std and quantiles of sampled pairwise distances), and (iii) spatial statistics capturing dispersion and local density (e.g., nearest-neighbor distance statistics).

BPP. We include: (i) scale features (number of items and bin capacity), and (ii) item-size distribution statistics (e.g., mean/std and quantiles), which summarize instance tightness and item-size heterogeneity.

MKP. We include: (i) scale features (number of items n and number of constraints m), and (ii) simple summary statistics of profits/weights and capacity tightness (e.g., mean/std over weights and the ratio between capacity and total weight per constraint).

D.2 Profile Grouping Methodology

We z-score normalize each dimension of $\phi(x)$ using statistics computed on $\mathcal{D}_{\text{train}}$. We then cluster the normalized profiles into G groups via K-means, yielding instance groups $\{\mathcal{D}_g\}_{g=1}^{G}$ with prototypes ϕ_g given by the corresponding centroids. These groups enable stratified mini-batch sampling during evolution and define the group-wise archives used by PLR. At test time, we assign a new instance to its nearest prototype (Eq. (9)) and retrieve the corresponding specialist.

D.3 Illustrative instance profiling and grouping on TSP

We use TSP as an illustrative example to show how instance profiles induce instance groups. We construct a small set of synthetic Euclidean TSP instances whose scale range and geometric heterogeneity are motivated by TSPLIB. Specifically, we generate 100 instances with the number of nodes $n \in [50, 1000]$. To cover heterogeneous spatial structures, we sample point sets from several common patterns, including near-uniform layouts, clustered mixtures, jitter grids, elongated rectangles, and ring-like layouts.

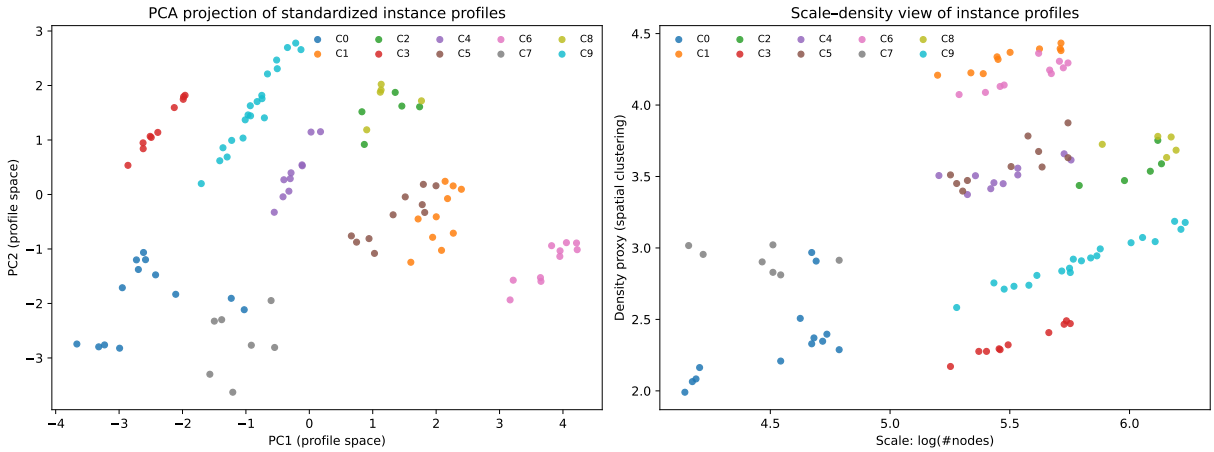

Figure 5: **Instance profiling and grouping (illustration on synthetic TSP instances).** We compute a profile vector for each instance (scale, distance statistics, spatial clustering, and shape descriptors), standardize the features, and apply k-means with G=10. *Left:* PCA projection of standardized profiles, colored by group assignment. *Right:* scale–density view using $\log(n)$ and a nearest-neighbor based density proxy.

G	Train gap (%)	Train time (s)	Test gap (%)	Test time (s)	Evaluation time per iteration (s)	Total Evaluation time (min)
5	0.550	2.050	0.620	1.950	28.1	46.7
10	0.419	1.905	0.442	1.775	35.0	58.3
15	0.410	1.915	0.438	1.780	50.5	83.3
20	0.408	1.925	0.437	1.785	65.7	108.3

Table 8: **Sensitivity to the number of instance groups G on TSP.** We vary the grouping granularity $G \in \{5, 10, 15, 20\}$ for DASH under the same evolution budget (100 evaluations, one solver per evaluation) while keeping the per-group sample size fixed (m instances per group), so the per-iteration batch size scales with G. Since the LLM generation budget is identical across G, we omit token counts and instead report evaluation time, which is the wall-clock time spent running candidate solvers on sampled instances.

Instance profile features. For each instance x with node coordinates $\{p_i\}_{i=1}^{n}$, we compute a lightweight profile vector $\phi(x)$ that summarizes scale, distance statistics, and spatial structure:

- **Scale:** $\log(n)$.
- **Distance distribution:** the coefficient of variation of (sampled) pairwise distances, and a robust spread statistic $\log\big(q_{90}(d)/q_{10}(d)\big)$, where $q_p(d)$ is the p-th quantile of pairwise distances.
- **Spatial clustering:** a density proxy comparing nearest-neighbor and typical pairwise distances, $\mathrm{dens}(x) = \log(\mu_{\mathrm{pair}}/\mu_{\mathrm{NN}})$, where μ_{NN} is the mean nearest-neighbor distance and μ_{pair} is the mean (sampled) pairwise distance. We also include the coefficient of variation of nearest-neighbor distances to capture heterogeneity.
- **Shape descriptors:** anisotropy measured by the square-root eigenvalue ratio of the coordinate covariance matrix, a log aspect ratio of the bounding box, and a radial dispersion statistic (the coefficient of variation of distances to the centroid).

All features are standardized by z-score normalization over the constructed set.

Grouping by K-means. We apply K-means to the standardized profile vectors $\phi(x)$ to obtain G instance groups $\{\mathcal{D}_g\}_{g=1}^{G}$. Figure 5 visualizes the resulting partition for G=10. The left panel shows a PCA projection of the standardized profiles, where each point is an instance and colors indicate its assigned group. The right panel plots two interpretable profile dimensions, scale $\log(n)$ and the density proxy $\mathrm{dens}(x)$, showing that groups differ in both size and spatial structure.

Sensitivity to the number of groups. The granularity G trades off finer stratification against the reliability and cost of group-wise evaluation under a fixed evolution budget. Table 8 reports results for $G \in \{5, 10, 15, 20\}$ under the same evolution

setting (100 evaluations, one solver per evaluation). We fix the per-group sample size m, so the per-iteration batch size scales with G. We report train and test gap and wall-clock time, and we also report the evaluation time spent running candidate solvers on sampled instances. Since the LLM generation budget is identical across G, token counts are omitted. Increasing G from 5 to 10 improves both train and test gaps. Further increasing G yields only marginal gap changes but substantially increases evaluation time, so we use $G = 10$ in the main experiments as a practical trade-off.

E Detailed DASH Workflow

We present Algorithm 1 for DASH and clarify its implementation details. All layer-wise acceptance conditions follow the tolerance-interval template and improvement threshold defined in Appendix F; prompt templates are given in Appendix J.

Notation. A solver is $\pi = (\theta, \sigma)$, where θ is the mechanism and σ is the schedule. In each iteration we sample per-group batches $\{\mathcal{B}_g\}_{g=1}^{G}$ with a *fixed* group batch size $|\mathcal{B}_g| = m$ for all g, and form the union batch $\mathcal{B} = \bigcup_{g=1}^{G} \mathcal{B}_g$, whose total size is $|\mathcal{B}| = Gm$. Evaluating a solver on $\mathcal{B}$ yields global batch-mean metrics $(\bar{\ell}, \bar{k}, \bar{t})$: terminal log-residual $\bar{\ell}$, trajectory efficiency $\bar{k}$, and runtime $\bar{t}$. We maintain (i) a global population $\mathcal{P}$ and (ii) group-wise archives $\{L_g\}_{g=1}^{G}$ for profiled library retrieval (PLR), where the retrieval library is the collection $\mathcal{L} = \{L_g\}_{g=1}^{G}$.

E.1 Initialization

We use a unified initialization template across tasks to ensure fair comparisons. Initialization instantiates (i) a seed solver $\pi_0 = (\theta_0, \sigma_0)$ for the corresponding backbone, and (ii) a deterministic evaluation setup so that parent–candidate comparisons are performed under identical per-instance initialization.

E.2 Explanation of the Workflow

Grouping and PLR. We compute an instance profile $\phi(x)$ for each training instance and cluster $\mathcal{D}$ into G groups $\{\mathcal{D}_g\}_{g=1}^{G}$ with prototypes $\{\phi_g\}$. At test time, we assign an instance x to its nearest prototype and deploy the best solver stored in the corresponding archive.

Per-iteration evolution. Each iteration selects a parent solver $\pi = (\theta, \sigma)$ and evaluates the parent and all candidates on the same union batch formed by sampling m instances from each group. MDL and MCL update the mechanism θ while keeping the schedule σ fixed; SSL updates the schedule σ while keeping the mechanism θ fixed (Compression then Enhancement). Accept and reject decisions follow Appendix F and always compare a candidate to its direct parent on the same batch.

Algorithm 1 DASH training with decoupled group-wise archiving and PLR inference

Inputs: task/backbone $\mathcal{T}$; training set $\mathcal{D}$; #groups G; iterations I; per-group batch size m; archive size K
Outputs: group archives $\{L_g\}_{g=1}^{G}$ and prototypes $\{\phi_g\}_{g=1}^{G}$

```
Compute profiles φ(x) and cluster D into groups {D_g}_{g=1}^G with prototypes {φ_g}
π_0 ← InitializeSolver(T)
Initialize population P ← {π_0} and archives L_g ← {π_0} for all g
for i = 1, ..., I do
    Select parent solver π from P
    Sample B_g ⊂ D_g with |B_g| = m for all g
    Evaluate π on the union batch ⋃_{g=1}^G B_g and record group-wise statistics on each B_g
    Initialize record set C ← {π}
    (π, C) ← LayerStep(MDL, π, {B_g}, C)
    (π, C) ← LayerStep(MCL, π, {B_g}, C)
    (π, C) ← LayerStep(SSL − 1, π, {B_g}, C)
    (π, C) ← LayerStep(SSL − 2, π, {B_g}, C)
    Update population P using the final solver π
    for each π̃ ∈ C do
        for g = 1, ..., G do
            Update L_g with π̃ using its recorded statistics on B_g; keep top-K
        end for
    end for
end for
PLR inference: for test instance x, compute φ(x), pick g* = arg min_g ||φ(x) − φ_g||_2, and return the best solver in L_{g*}
```

Decoupled group-wise archiving. In each iteration, we record group-wise evaluation statistics for every solver evaluated during three layers' iteration, regardless of whether it is accepted into the global population. These recorded statistics are then used to update each group archive L_g (keeping top-K), without additional solver runs. In our implementation, each L_g keeps K solvers with the lowest group-wise terminal log-residual $\bar{\ell}_g$ measured on $\mathcal{B}_g$ under same evaluation protocol. When multiple solvers have the same $\bar{\ell}_g$, we compare them using the recorded trajectory score $\bar{k}_g$ and runtime $\bar{t}_g$.

F Acceptance Rules and Thresholds

We use a unified acceptance protocol based on batch-averaged metrics: terminal log-residual $\bar{\ell}$, trajectory score $\bar{k}$ (tLDR), and runtime $\bar{t}$. Lower is

better for $\bar{\ell}$ and $\bar{t}$; higher is better for $\bar{k}$. All decisions compare a candidate against its direct parent on the same evaluation batch.

Tolerance. We say a metric x' is within tolerance of its parent value x if:

$$\begin{aligned} |x' - x| &\le \epsilon \quad \text{for } x = \bar{\ell}, \\ |x' - x| &\le \epsilon \cdot |x| \quad \text{for } x \in \{\bar{k}, \bar{t}\}. \end{aligned} \tag{17}$$

Improvement. We define x' is an improvement over x if:

$$\bar{\ell}' \le \bar{\ell} - \delta, \qquad \bar{t}' \le (1-\delta)\bar{t}, \qquad \bar{k}' \ge (1+\delta)\bar{k}. \tag{18}$$

MDL (Mechanism Discovery). *Constraint:* update θ with σ fixed; accept by terminal-quality improvement, breaking ties by trajectory score.

$$\begin{aligned} \mathrm{Accept}(\theta') \iff & \big(\bar{\ell}' \le \bar{\ell} - \delta\big) \\ \vee & \big(|\bar{\ell}' - \bar{\ell}| \le \epsilon \wedge \bar{k}' \ge (1+\delta)\bar{k}\big). \end{aligned} \tag{19}$$

MCL (Mechanism Consolidation). *Constraint:* refactor θ while preserving behavior; accept only if both $\bar{\ell}$ and $\bar{k}$ remain within tolerance.

$$\mathrm{Accept}(\theta') \iff \big(|\bar{\ell}' - \bar{\ell}| \le \epsilon\big) \wedge \big(|\bar{k}' - \bar{k}| \le \epsilon \cdot |\bar{k}|\big). \tag{20}$$

SSL-1 (Compression). *Constraint:* update σ with θ fixed; accept by reducing runtime while keeping terminal quality within tolerance.

$$\mathrm{Accept}(\sigma') \iff \big(\bar{t}' \le (1-\delta)\bar{t}\big) \wedge \big(|\bar{\ell}' - \bar{\ell}| \le \epsilon\big). \tag{21}$$

SSL-2 (Enhancement). *Constraint:* update σ with θ fixed under the same runtime budget as the parent; accept by improving terminal quality while keeping trajectory score within tolerance.

$$\mathrm{Accept}(\sigma') \iff \big(\bar{\ell}' \le \bar{\ell} - \delta\big) \wedge \big(|\bar{k}' - \bar{k}| \le \epsilon \cdot |\bar{k}|\big). \tag{22}$$

G Mechanism and Schedule in GLS–TSP

We specify the mechanism θ and the schedule σ in GLS–TSP, describe the edit interfaces of MDL/MCL/SSL, and provide a DASH-evolved GLS solver example on TSPLIB.

G.1 Mechanism and Schedule details in GLS–TSP

Table 9 summarizes what we treat as mechanism choices (θ) and schedule controls (σ) in GLS–TSP.

Mechanism θ. The mechanism θ specifies the transition rules of the search, including the neighborhood restriction, move operators and scan strategy, acceptance rule, perturbation design (if any), and the guidance rule (how penalties/guidance signals are computed and updated).

Schedule σ. The schedule σ specifies runtime control: when to invoke each mechanism component and how long to run it, including phase budgets, stopping rules, and trigger conditions. θ defines the available transitions; σ controls their timing and allocation.

G.2 Solver example on TSPLIB

We provide a TSPLIB case study on case *a280*. Given the instance profile, PLR retrieves a group-specific GLS solver from the archive. The retrieved solver reaches the optimum (Gap $= 0\%$) and terminates early in ≈ 0.48s under a 4s time budget.

What DASH changes across layers. Starting from the task initialization, DASH modifies the solver through the same layer functions as in the main workflow (illustrated in Figure 6 and Figure 7):

- **MDL (Mechanism Discovery).** Edits the search components that define state transitions, including the kNN candidate restriction (k=16), first-improvement 2-opt, improve-only acceptance, and a bounded 2-opt kick perturbation.
- **MCL (Mechanism Consolidation).** Refactors the guidance implementation while preserving its behavior; the solver keeps a GLS-style penalty update based on edge utilities.
- **SSL-1 (Compression).** Compresses runtime by tightening schedule budgets (e.g., a 4s cap and a reduced loop budget of 150), without sacrificing terminal quality under the acceptance rule.
- **SSL-2 (Enhancement).** Adjusts a single runtime-control parameter (the stagnation threshold `max_no_improve`=30) to balance continued search and early termination within the same compressed cap.

On *a280*, the solver reaches the optimum quickly and then triggers the stagnation-based stopping rule, yielding ≈ 0.48s runtime. This example illustrates that DASH can jointly obtain high-quality search logic and tight runtime control within one evolutionary process, while PLR enables selecting an instance-matched specialist without additional LLM calls.

Component	Search Mechanism θ	Runtime Schedule σ
Initialization	tour construction (fixed for fairness)	(none; fixed single start)
Neighborhood	candidate restriction and move neighborhood definition	when to update / switch
Local improvement	move operators and acceptance logic	phase budgets and iteration/stop caps
Guidance	guidance/penalty rule and how it interacts with search	when to activate and how often to update
Perturbation	perturbation operator	triggering/frequency and (optional) intensity
Stopping	(none)	time cap and stagnation-based stopping

Table 9: Search Mechanism (θ) and Runtime Shedule (σ) in GLS–TSP.

Method	TSPLIB-small		TSPLIB-large	
	Gap(%)	Time(s)	Gap(%)	Time(s)
FunSearch	0.152	12.343	1.146	289.012
ReEvo	0.147	4.560	2.032	353.450
EoH	0.097	13.403	1.138	476.502
MEoH	0.189	2.786	1.637	109.059
Hercules	0.096	9.458	1.391	594.495
DASH	0.122	0.263	1.024	4.520

Table 10: **TSPLIB results under time-unconstrained evaluation.** We report Gap (%) and Time (s) on TSPLIB-small (14–200) and TSPLIB-large (201–1000).

Method	TSPLIB-small		TSPLIB-large	
	Gap(%)	Time(s)	Gap(%)	Time(s)
FunSearch	0.155	10.211	3.344	61.591
ReEvo	0.134	10.406	3.057	62.026
EoH	0.136	10.723	5.014	63.536
MEoH	0.094	9.801	3.066	60.844
Hercules	0.112	10.345	2.703	62.203
DASH	0.010	9.636	0.901	59.442

Table 11: **TSPLIB performance under time-matched evaluation.** We fix the Time budget to T=10s for TSPLIB-small (14–200) and T=60s for TSPLIB-large (201–1000). For DASH, we disable SSL-1 to prevent runtime compression.

H Comparisons Under Time-Matched and Unconstrained Protocols

Table 10 evaluates the learned solvers under unconstrained time budget, reporting the wall-clock runtime until each solver terminates by its own stopping rules. On TSPLIB-large, prior LHD baselines often require 10^2–10^3s to reach around 1%–2% gaps, while DASH attains a comparable gap (1.024%) in 4.52s, indicating that the co-evolved schedule effectively prunes low-yield computation and eliminates substantial computational slack. Table 11 further controls for runtime by enforcing fixed budgets (10s for TSPLIB-small and 60s for TSPLIB-large) and disabling SSL-1 for DASH to avoid runtime compression. Even under equal budgets, DASH achieves smaller gaps (0.010% / 0.901%) than all baselines, suggesting that its advantage is not solely due to stopping earlier but also due to improved mechanisms and trajectory-aware selection that accelerate convergence within the same budget.

Table 12 reports additional comparisons on TSPLIB against Concorde, LKH3, and Google OR-Tools. OR-Tools does not provide trajectory-aware early stopping in our setting, so we run it with a fixed time limit of 60 s per instance and report its final solution at timeout. To compare efficiency under the same runtime, we further evaluate OR-Tools with an instance-wise time limit matched to the runtime of DASH on the same instance (OR-Tools*). On average, OR-Tools under 60 s attains a higher gap than DASH (1.406% vs. 0.442%) while using much more runtime (60.051 s vs. 1.775 s). Under the time-matched protocol, ORTools* yields substantially worse gaps (8.025%), and it fails to return a solution on some instances because the matched time limit is too small for OR-Tools to produce a result. These results indicate that DASH reaches high-quality solutions with less runtime than a dedicated solver.

I Benchmark Problems and Solver Backbones

We summarize the benchmark problems and standardize the solution representation, objective, and gap computation for each task (Table 13), and list the solver backbones and referenced packages used in our experiments (Table 14).

J Prompt Template

We summarize the prompt templates used by DASH for the three iteration layers: MDL/M-CL/SSL in Figure 8–11.

Case study (a280 specialist): Mechanism θ

```
def Initialization(coords, start=0):
    # init is fixed for fairness (start=0)
    tour = nearest_neighbor_tour(coords, start=start)
    return tour

def Neighborhood(tour, coords, k=16):
    # kNN candidate restriction  [MDL]
    knn = knn_lists(coords, k=k)
    moves = two_opt_candidates(tour, knn)
    return moves

def LocalImprovement(tour, moves, strategy="first"):
    # 2-opt, first-improvement  [MDL]
    for (i, j) in moves:
        if delta_2opt(tour, i, j) < 0:
            return apply_2opt(tour, i, j), True
    return tour, False

def Guidance(tour, dist, penalty, top_k=4, gls_lambda=1.2,
             weight=1.0, lam=0.5):
    # baseline GLS penalty rule (kept)  [MCL]
    util = edge_utilities(tour, dist, penalty, lam=lam)
    edges = select_topk_edges(util, k=top_k)
    for e in edges:
        penalty[e] += weight
    return penalty, gls_lambda

def Perturbation(tour, rng, strength=1):
    # bounded kick  [MDL]
    for _ in range(strength):
        i, j = rng.randint(0, len(tour)-1), rng.randint(0, len(tour)-1)
        if i > j: i, j = j, i
        tour = apply_2opt(tour, i, j)
    return tour

def Accept(curr_cost, cand_cost):
    # improve-only
    return cand_cost < curr_cost
```

Figure 6: Mechanism (θ) of the a280 specialist. Inline tags indicate typical MDL/MCL edit locations.

Case study (a280 specialist): Schedule σ

```
def RunSchedule(instance,
                time_limit_s=4.0,     # [SSL-1] compressed cap (kept)
                loop_max=150,         # [SSL-1] reduced loop budget (kept)
                max_no_improve=30,    # [SSL-2] single-field enhancement
                k=16, top_k=4, gls_lambda=1.2,
                weight=1.0, lam=0.5, kick_strength=1):
    t0 = now()
    best = Initialization(instance.coords, start=0)
    best_cost = tour_length(best, instance.dist)
    penalty = init_penalty_matrix(instance.n)
    no_imp = 0

    for it in range(loop_max):
        if now() - t0 > time_limit_s: break
        if no_imp >= max_no_improve: break  # [SSL-2] early-stop threshold

        moves = Neighborhood(best, instance.coords, k=k)
        cand, improved = LocalImprovement(best, moves, strategy="first")
        cand_cost = tour_length(cand, instance.dist)

        if improved and Accept(best_cost, cand_cost):
            best, best_cost = cand, cand_cost
            no_imp = 0
        else:
            no_imp += 1

        penalty, _ = Guidance(best, instance.dist, penalty,
                              top_k=top_k, gls_lambda=gls_lambda,
                              weight=weight, lam=lam)

        if no_imp > 0 and (no_imp % 10 == 0):
            best = Perturbation(best, instance.rng, strength=kick_strength)

    return best
```

Figure 7: Schedule (σ) of the a280 specialist. Inline tags indicate typical SSL-1/SSL-2 edit locations.

TSPLIB	n	Concorde		LKH3		ORTools		ORTools*		DASH	
		Gap(%)	Time(s)	Gap(%)	Time(s)	Gap(%)	Time(s)	Gap(%)	Time(s)	Gap(%)	Time(s)
burma14	14	0.000%	0.060	0.000%	0.005	0.000%	60.007	0.000%	0.160	0.000%	0.160
ulysses16	16	0.000%	0.220	0.000%	0.007	0.000%	60.009	0.000%	0.114	0.000%	0.114
gr17	17	0.000%	0.080	0.000%	0.009	0.000%	60.025	0.000%	0.099	0.000%	0.099
gr21	21	0.000%	0.030	0.000%	0.008	0.000%	60.029	0.000%	0.198	0.000%	0.198
ulysses22	22	0.000%	0.530	0.000%	0.023	0.000%	60.002	0.000%	0.210	0.000%	0.210
gr24	24	0.000%	0.070	0.000%	0.014	0.000%	60.007	0.000%	0.304	0.000%	0.304
fri26	26	0.000%	0.070	0.000%	0.012	0.000%	60.003	0.000%	0.285	0.000%	0.285
bayg29	29	0.000%	0.090	0.000%	0.017	0.000%	60.004	6.522%	0.200	0.000%	0.200
bays29	29	0.000%	0.130	0.000%	0.022	0.000%	60.012	8.453%	0.114	0.000%	0.114
dantzig42	42	0.000%	0.230	0.000%	0.040	0.000%	60.020	0.000%	0.128	0.000%	0.128
swiss42	42	0.000%	0.130	0.000%	0.041	0.000%	60.014	0.000%	0.044	0.000%	0.044
att48	48	0.000%	0.560	0.000%	0.030	0.790%	61.275	6.869%	0.098	0.000%	0.098
gr48	48	0.000%	0.670	0.000%	0.036	0.000%	60.017	0.000%	0.053	0.000%	0.053
hk48	48	0.000%	0.170	0.000%	0.041	0.000%	60.013	0.000%	0.326	0.000%	0.326
eil51	51	0.000%	0.730	0.000%	0.032	0.000%	60.006	0.000%	0.308	0.000%	0.308
berlin52	52	0.000%	0.290	0.000%	0.038	0.000%	60.006	0.000%	0.032	0.000%	0.032
brazil58	58	0.000%	0.680	0.000%	0.039	0.000%	60.006	0.000%	0.028	0.000%	0.028
st70	70	0.000%	0.500	0.000%	0.041	0.000%	60.004	-	-	0.000%	0.067
eil76	76	0.000%	0.110	0.000%	0.055	0.000%	60.006	0.000%	0.054	0.000%	0.054
pr76	76	0.000%	0.600	0.000%	0.056	0.000%	60.007	0.000%	0.130	0.000%	0.130
gr96	96	0.000%	0.840	0.000%	0.084	0.000%	60.008	-	-	0.205%	0.108
rat99	99	0.000%	0.400	0.000%	0.092	0.000%	60.005	0.000%	0.325	0.000%	0.325
kroA100	100	0.000%	0.310	0.000%	0.113	0.000%	60.004	0.000%	0.153	0.000%	0.153
kroB100	100	0.000%	0.580	0.000%	0.103	0.000%	60.004	0.000%	0.164	0.000%	0.164
kroC100	100	0.000%	0.830	0.000%	0.078	0.000%	60.004	0.000%	0.158	0.000%	0.158
kroD100	100	0.000%	0.750	0.000%	0.089	0.000%	60.004	0.000%	0.178	0.000%	0.178
kroE100	100	0.000%	0.990	0.000%	0.084	0.000%	60.004	0.000%	0.066	0.172%	0.066
rd100	100	0.000%	0.670	0.000%	0.105	0.000%	60.004	0.000%	0.114	0.000%	0.114
eil101	101	0.000%	0.740	0.000%	0.102	0.000%	60.004	0.000%	0.305	0.000%	0.305
lin105	105	0.000%	0.590	0.000%	0.120	0.000%	60.005	0.000%	0.240	0.000%	0.240
pr107	107	0.000%	1.030	0.000%	0.105	0.000%	60.006	0.000%	0.787	0.000%	0.787
gr120	120	0.000%	2.230	0.000%	0.137	0.000%	60.006	1.195%	0.785	0.130%	0.785
pr124	124	0.000%	3.640	0.000%	0.189	0.000%	60.009	0.000%	0.272	0.000%	0.272
bier127	127	0.000%	1.650	0.000%	0.104	0.000%	60.008	11.788%	0.334	0.084%	0.334
ch130	130	0.000%	2.130	0.000%	0.115	0.000%	60.006	8.361%	0.241	0.475%	0.241
pr136	136	0.000%	3.970	0.000%	0.148	0.000%	60.006	-	-	0.105%	0.090
gr137	137	0.000%	3.420	0.000%	0.113	0.000%	60.005	-	-	0.399%	0.122
pr144	144	0.000%	2.580	0.000%	0.138	0.000%	60.007	-	-	0.108%	0.131
ch150	150	0.000%	3.030	0.000%	0.155	0.000%	60.006	-	-	0.352%	0.193
kroA150	150	0.000%	5.000	0.000%	0.148	0.000%	60.005	10.849%	0.397	1.078%	0.397
kroB150	150	0.000%	4.230	0.000%	0.188	0.000%	60.005	-	-	0.896%	0.303
pr152	152	0.000%	7.930	0.000%	0.146	0.000%	60.007	-	-	0.185%	0.115
u159	159	0.000%	1.000	0.000%	0.211	0.000%	60.006	0.000%	0.365	0.000%	0.365
si175	175	0.000%	13.090	0.000%	0.229	0.000%	60.007	3.511%	0.211	0.187%	0.211
brg180	180	0.000%	1.460	0.000%	0.231	0.000%	60.008	0.000%	1.945	0.000%	1.945
rat195	195	0.000%	22.230	0.000%	0.280	0.000%	60.009	-	-	0.517%	0.193
d198	198	0.000%	11.820	0.000%	0.285	0.000%	60.010	-	-	0.475%	0.227
kroA200	200	0.000%	6.590	0.000%	0.225	0.000%	60.012	7.691%	0.996	0.466%	0.996
kroB200	200	0.000%	3.910	0.000%	0.177	0.000%	60.011	6.414%	0.424	0.132%	0.424
gr202	202	0.000%	5.010	0.000%	0.258	0.000%	60.013	4.430%	1.144	2.470%	1.144
ts225	225	0.000%	20.520	0.000%	0.272	0.000%	60.014	0.000%	0.989	0.000%	0.989
tsp225	225	0.000%	15.010	0.000%	0.367	0.000%	60.014	8.634%	4.202	1.634%	4.202
pr226	226	0.000%	4.350	0.000%	0.302	0.000%	60.015	11.057%	0.627	0.056%	0.627
gr229	229	0.000%	38.610	0.000%	0.254	0.000%	60.015	-	-	0.541%	0.129
gil262	262	0.000%	13.060	0.000%	0.436	0.000%	60.017	18.930%	1.440	2.313%	1.440
pr264	264	0.000%	2.670	0.000%	0.235	0.000%	60.017	-	-	0.000%	0.445
a280	280	0.000%	5.370	0.000%	0.151	1.939%	61.358	21.365%	0.483	0.000%	0.483
pr299	299	0.000%	17.490	0.000%	0.924	0.435%	60.019	23.341%	1.228	0.708%	1.228
lin318	318	0.000%	9.740	0.000%	0.589	0.000%	60.020	-	-	1.511%	0.928
rd400	400	0.000%	148.420	0.000%	0.711	0.000%	60.022	19.475%	3.110	0.340%	3.110
fl417	417	0.000%	57.750	0.000%	3.982	2.949%	60.023	19.019%	6.143	1.703%	6.143
gr431	431	0.000%	133.290	0.000%	0.836	4.472%	60.025	-	-	1.338%	1.437
pr439	439	0.000%	216.750	0.000%	1.758	7.985%	60.024	-	-	3.126%	1.330
pcb442	442	0.000%	49.920	0.000%	2.538	1.103%	60.024	11.184%	5.701	0.532%	5.701
d493	493	0.000%	113.320	0.000%	26.361	3.904%	60.026	19.242%	3.694	1.920%	3.694
att532	532	0.000%	109.520	0.000%	4.183	5.736%	60.150	28.661%	5.692	0.943%	5.692
ali535	535	0.000%	53.140	0.000%	1.276	8.553%	60.062	25.106%	3.511	0.784%	3.511
si535	535	0.000%	43.130	0.000%	35.106	5.315%	60.029	16.304%	9.588	0.553%	9.588
pa561	561	0.000%	246.820	0.000%	4.992	4.206%	60.033	20.733%	4.956	0.724%	4.956
u574	574	0.000%	23.040	0.000%	6.342	8.105%	60.082	26.410%	4.970	1.284%	4.970
rat575	575	0.000%	363.070	0.000%	6.103	6.407%	60.037	18.918%	5.111	0.930%	5.111
p654	654	0.000%	26.520	0.000%	10.456	5.127%	60.020	-	-	0.341%	3.302
d657	657	0.000%	260.370	0.002%	17.131	9.562%	60.034	32.799%	7.911	1.045%	7.911
gr666	666	0.000%	49.860	0.000%	23.884	10.749%	60.078	28.823%	16.217	1.199%	16.217
u724	724	0.000%	225.440	0.000%	4.058	8.533%	60.020	27.615%	17.717	0.826%	17.717
rat783	783	0.000%	37.880	0.000%	0.659	10.970%	60.024	27.811%	8.462	0.818%	8.462
Avg.	-	0.000%	31.627	0.000%	2.092	1.406%	60.051	8.025%	2.097	0.442%	1.775

Table 12: TSPLIB per-instance results. ORTools* uses time budgets matched to DASH; "-" indicates no solution under the budget.

Problem	Problem description
TSP	**Traveling Salesman Problem (Euclidean).** Given n nodes with coordinates $\{p_i\}_{i=1}^n$, find a Hamiltonian cycle visiting each node exactly once. **Output:** a permutation (tour) over nodes. **Objective:** minimize total tour length (sum of Euclidean edge distances). **Evaluation:** we evaluate a solution by its tour length f (lower is better) and compute the optimality gap using the reference $f^\star$. **Reference optimum/best-known ($f^\star$):** for benchmark instances, we use the best-known/optimal tour lengths reported by the corresponding benchmark suite (e.g., TSPLIB).
CVRP	**Capacitated Vehicle Routing Problem (Euclidean).** Given a depot, a set of customers with demands, and vehicle capacity Q, find a set of routes. **Constraints:** each route starts/ends at the depot; each customer is served exactly once; total demand on each route $\leq Q$. **Objective:** minimize total travel cost (sum of route lengths). **Evaluation:** we evaluate a solution by its total route length f (lower is better) and compute the optimality gap using the reference $f^\star$. **Reference optimum/best-known ($f^\star$):** optimal/best-known values from CVRPLIB.
BPP	**Online Bin Packing Problem.** Given a stream of item sizes $\{w_t\}_{t=1}^m$ and bins of capacity C, items arrive sequentially and must be assigned irrevocably upon arrival. **Output:** an online assignment policy that maps each arriving item to an existing bin or to a newly opened bin. **Constraints:** for each bin, the sum of assigned item sizes $\leq C$ at all times. **Objective:** minimize the number of bins used after processing the full stream. **Evaluation:** we evaluate a run by the final number of bins f (lower is better) and compute the optimality gap using the reference $f^\star$. **Reference optimum/best-known ($f^\star$):** the relaxation lower bound lb computed for each instance (fixed item stream), used as the reference for gap computation.
MKP	**Multi-dimensional Knapsack Problem.** Given items with profit p_i and resource consumptions $\{a_{ij}\}$ under d constraints with capacities $\{b_j\}$, select a subset of items. **Constraints:** $\sum_i a_{ij}x_i \leq b_j$ for all $j = 1, \ldots, d$, with $x_i \in \{0, 1\}$. **Objective:** maximize total profit $\sum_i p_i x_i$. **Evaluation:** we report gap under a minimization convention by evaluating $f = -\sum_i p_i x_i$ (equivalently maximizing profit). **Reference optimum/best-known ($f^\star$):** optimal values from OR-Library.

Table 13: Problem definitions used in our experiments.

Solver / Package	Solver description
GLS	**Guided Local Search (GLS).** A local-search backbone augmented with an adaptive penalty mechanism that discourages repeatedly using "undesirable" components (e.g., edges) to escape local minima. **Role in our paper:** primary TSP backbone (fixed evaluation protocol / time budget).
ACO	**Ant Colony Optimization (ACO).** A population-based constructive metaheuristic: ants build solutions guided by pheromone scores and problem heuristics; pheromones are updated based on solution quality to bias future construction. **Role in our paper:** unified backbone for CVRP/MKP to keep a consistent solver skeleton across methods.
GOA	**Greedy Online Assignment (GOA).** An online constructive heuristic for bin packing: items arrive sequentially and are irrevocably assigned to an existing bin (based on a bin-scoring rule) or to a newly opened bin when necessary. **Role in our paper:** online BPP backbone following the standard greedy assignment protocol (fixed item stream / evaluation protocol).
ILS	**Iterated Local Search (ILS).** Alternates between (i) local improvement and (ii) perturbation / restart to explore new basins of attraction, typically keeping the best-so-far incumbent. **Role in our paper:** alternative TSP backbone used to study cross-framework transferability.
LKH (Python approx.)	**Lin–Kernighan(-Helsgaun) style heuristic (approximate Python implementation).** A variable-depth k-opt search paradigm for TSP; our implementation is a Python-level approximation used for controlled transfer experiments. **Role in our paper:** alternative TSP backbone (explicitly noted as Python-implemented approximation).
Concorde	**Concorde TSP Solver (official package).** A classical high-performance (exact/near-exact) TSP solver used as a strong off-the-shelf baseline / reference point for solution quality.
LKH3	**LKH3 (official package).** The official LKH implementation used as a classical TSP heuristic baseline with strong performance and engineering optimizations.
OR-Tools	**Google OR-Tools (official package).** A widely used operations-research toolkit providing mature routing/assignment solvers (e.g., TSP).

Table 14: Solver backbones and referenced solver packages.

{TASK_PROMPT}

I have K existing algorithms (each includes a 1-sentence idea + code) and their evaluation results:

[Algorithm 1]
Idea: {IDEA_1}
Gap (%): {GAP_1} Time (s): {TIME_1}
Code:
{CODE_1}
...
[Algorithm K]
Idea: {IDEA_K}
Gap (%): {GAP_K} Time (s): {TIME_K}
Code:
{CODE_K}

Your task (MDL = mechanism discovery inspired by history):

- Write a NEW algorithm that absorbs useful patterns from these algorithms, but is not a copy.
- Target: lower Gap than the best provided one, while avoiding a large increase in Time.

Output requirements:

1. First, write ONE sentence inside { } describing:
   - what you learned from the parents, AND
   - what key change you introduce in the new algorithm.
2. Then output ONLY valid Python code implementing the function:
   {FUNC_NAME}({INPUTS}) → {OUTPUTS}
   Keep input/output shapes and names unchanged.
   {INOUT_INF} {OTHER_INF}

Do not provide any extra explanations.

Figure 8: The prompt template for the MDL (Mechanism Discovery).

{TASK_PROMPT}

I have K existing algorithms (idea + code) and their evaluation results:

[Algorithm 1]
Idea: {IDEA_1}
Gap (%): {GAP_1} Time (s): {TIME_1}
Code:
{CODE_1}

...

[Algorithm K]
Idea: {IDEA_K}
Gap (%): {GAP_K} Time (s): {TIME_K}
Code:
{CODE_K}

Your task (MCL = consolidation):

- Do NOT significantly increase Time or Gap compared with the best trade-off among the K algorithms.
- Merge duplicated branches, remove dead/unnecessary logic, unify style/interface.
- Allow light parameter retuning; avoid heavy computations.
- Avoid brittle instance-specific constants.

Output requirements:

1. First, write ONE sentence inside { } describing:
   - what redundancy you remove, AND
   - what behavior you preserve.
2. Then output ONLY valid Python code implementing:
   {FUNC_NAME}({INPUTS}) → {OUTPUTS}
   Keep input/output shapes unchanged.
   {INOUT_INF} {OTHER_INF}

Do not provide any extra explanations.

Figure 9: The prompt template for the MCL (Consolidation).

{TASK_PROMPT}

You are editing the solver SCHEDULE (runtime controls).
Do NOT change the mechanism code; only change schedule/spec fields.

Current performance:
Gap (%): {GAP_PARENT} Time (s): {TIME_PARENT}

Trace summary:
{TRACE_SUMMARY}

Current ScheduleSpec (JSON):
{PARENT_SCHEDULE_JSON}

Your task (SSL-stage-1 = compression):

- Reduce Time by editing schedule controls (budgets, stopping rules, triggers).
- Keep Gap not worse (or as close as possible).
- Make minimal changes and avoid obvious compute increases.

Output requirements:

- Output ONLY a valid JSON dict (no extra text).
- Keep keys/types consistent with the given schema.

Figure 10: The prompt template used for the SSL-stage-1.

{TASK_PROMPT}

You are editing the solver SCHEDULE (runtime controls).
This is SSL-stage-2: change exactly ONE schedule control field to improve quality.

Current performance:
Gap (%): {GAP_PARENT} Time (s): {TIME_PARENT}

Trace summary:
{TRACE_SUMMARY}

Current ScheduleSpec (JSON):
{PARENT_SCHEDULE_JSON}

Your task (SSL-stage-2 = single-field enhancement):

- Change exactly ONE field in the provided ScheduleSpec (choose one from {ALLOWED_CONTROL_FIELD_LIST}).
- Goal: reduce Gap without increasing overall runtime (do not increase budgets/caps).
- Keep the rest unchanged (minimal diff).

Output requirements:

- Output ONLY a valid JSON dict (no extra text).
- Keep schema keys/types consistent.
- Minimal diff outside the changed field.

Figure 11: The prompt template used for the SSL-stage-2.