AUTOMATING COMBINATORIAL OPTIMIZATION KNOWLEDGE TRANSFER WITH LLMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Developing combinatorial optimization algorithms is time-consuming and labor-intensive. While recent LLM-based methods enable automated algorithm design, they often overlook the transferable knowledge encoded in mature solvers. In this paper, we study *Automated Knowledge Transfer (AutoKT)*, which asks: *to what extent can LLMs automate knowledge transfer from a source solver to a target problem?* To this end, we formalize the algorithmic knowledge space and transfer process, defining a comprehensive evaluation protocol across three tiers: same-domain, cross-domain, and new-domain transfer. We then develop a three-stage agentic framework: a local migration stage that incrementally constructs a scaffold from localized source knowledge, a global migration stage that coordinates the transferred knowledge, and a validation stage to ensure correctness. Extensive experiments demonstrate AutoKT's effectiveness across all three tiers, outperforming both strong domain heuristics and general-purpose coding agents. We further provide in-depth analyses revealing key insights into AutoKT's behavioral patterns and knowledge transferability, offering useful takeaways for the community.

1 INTRODUCTION

Combinatorial optimization (CO) drives wide applications such as logistics (Sbihi and Eglese, 2010; Song et al., 2025), scheduling (Zhao et al., 2020; Gannouni et al., 2020; Shobaki et al., 2013), and resource allocation (Skobelev, 2011; Nansheng and Qichen, 2023; Nearchou and Omirou, 2006). Decades of research have produced high-performance solvers, yet reusing the knowledge encoded in them is non-trivial. State-of-the-art solvers are highly domain-specific, and even minor changes to constraints or objectives can break much of their logic. As a result, identifying reusable knowledge and developing the missing components for a new problem remains labor-intensive and demands substantial engineering expertise.

Recently, large language models (LLMs) have opened new possibilities for automated algorithm discovery through their agentic reasoning and code-generation capabilities (Ye et al., 2024; Liu et al., 2024a; Jiang et al., 2025; Zhang et al., 2025a; Romera-Paredes et al., 2024). Existing efforts either refine predefined operators or heuristics for a single target problem (Ye et al., 2024; Liu et al., 2024a; Romera-Paredes et al., 2024), or generate new solvers from scratch (Li et al., 2025a; Zhang et al., 2025a). Both paradigms treat each problem in isolation and overlook knowledge transfer across problems, leaving the knowledge embedded in decades of mature solvers largely untapped.

This paper therefore asks a different research question: *Given a mature solver, to what extent can LLMs transfer its knowledge to derive specialized solvers for new CO problems?* Answering it is challenging: knowledge must be properly represented, transferred systematically, and kept consistent despite complex dependencies among its components. To the best of our knowledge, no existing work has addressed this question, leaving the capability of LLMs for solver knowledge transfer unknown.

To this end, we introduce a new paradigm, termed **Automated Knowledge Transfer (AutoKT)**. AutoKT features a three-stage agentic framework. First, to make knowledge explicit and transferable, the *local migration* stage decomposes the source solver into three aspects, *problem*, *algorithm design*, and *implementation*, and distills them into two technical reports: a *summary* report capturing the source's algorithmic knowledge, and an *adaptation* report justifying whether each component is *mapped*, *replaced*, or *dropped* based on its compatibility with the target. Migration then proceeds incrementally via a plan–transfer–evaluate–reflect loop, guarded by a completeness checklist against

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.

the LLM's tendency to leave to-dos unfinished. In one-to-many transfers, an experience pool accumulates new knowledge discovered online and reuses it to guide subsequent transfers. Next, to keep the resulting algorithm coherent and complete, the *global migration* stage revisits the transferred solver skeleton from a global perspective, jointly coordinating the roles and interactions of its components. By refining each component in the context of all transferred knowledge, it reduces interference arising from component-wise migration and improves overall solver performance. Finally, the *validation* stage ensures feasibility through automated unit testing.

To ensure systematic evaluation, we introduce an evaluation protocol for AutoKT spanning three tiers of difficulty: *same-domain*, *cross-domain*, and *new-domain* transfer. AutoKT surpasses HGS-PyVRP (Wouda et al., 2024) on 15 VRP variants in *same-domain transfer*, outperforms OR-Tools on JSSP and achieves competitive performance on FJSP in *cross-domain transfer*, and, in *new-domain transfer*, constructs effective heuristic solvers for CPMP and BPP, matching the reference optimum on CPMP while achieving competitive solution quality on BPP. Beyond performance, our in-depth analyses shed light on AutoKT's behavior and reveal several insights into knowledge transferability. Finally, we highlight a broader meta-science takeaway: AutoKT enables rapid baseline construction, producing competitive reference solvers within minutes in the evaluated settings.

Contributions: (i) we introduce and formalize a new task called *AutoKT*, an LLM-driven paradigm for systematically transferring knowledge from mature CO solvers, together with a three-tier transferability protocol; (ii) we propose a three-stage agentic framework that effectively operationalizes AutoKT; (iii) AutoKT matches or outperforms established baselines across all transfer tiers, and constructs functional heuristic solvers for previously unseen target problems while revealing that generl-perper coding agen tis not suffient for thi task and (iv)) our analyses further examine the effectiveness and trade-offs of different transfer strategies, as well as knowledge transferability and the broader implications of AutoKT, offering practical insights for the community.

2 Related Work

Algorithms for COPs. Extensive research has designed high-performance heuristics for specific COPs. For example, SISR (Christiaens and Vanden Berghe, 2020) and FILO (Accorsi and Vigo, 2021) are powerful large neighborhood search methods for CVRP, and Li et al. (2025b) combine hash-based tabu search with frequent pattern mining for JSSP. Although problem-specific, these methods embed reusable algorithmic knowledge, such as local search, exploration-exploitation balancing, and neighborhood design, that can generalize across COP solvers. Automating the transfer of such knowledge, however, remains an open challenge.

Transfer learning for COPs. Recent neural approaches explore multi-task and foundation-style models that share parameters across problem variants, such as RouteFinder (Berto et al., 2024), MVMoE (Zhou et al., 2024), and MTL-POMO (Liu et al., 2024b), showing promising results on multiple routing variants through large-scale pretraining. However, these approaches remain at an early stage: they generalize poorly, incur high training costs (typically 10^6–10^7 training instances and multi-day GPU training), and still lag behind state-of-the-art HUMAN heuristics such as HGS-PyVRP.

Coding agents and automated heuristic design (AHD). Coding agents such as Claude Code, Codex, and OpenCode have demonstrated strong capabilities in scaling LLMs to repository-level software engineering tasks (Jimenez et al., 2023). However, they lack systematic mechanisms and domain-specific expertise for effective knowledge transfer across optimization problems. In AHD, EOH (Liu et al., 2024a) and ReEvo (Ye et al., 2024) perform LLM-guided search over heuristic components, ARS (Li et al., 2025a) extends this paradigm to multiple problems via operator search, and AFL (Zhang et al., 2025a) generates solvers through multi-agent pipelines. However, these approaches either operate on problem-specific operators or synthesize solvers from scratch, overlooking the structural knowledge embedded in existing algorithms and thereby limiting transfer efficiency and applicability. Our framework is designed to address both limitations.

3 Problem Formulation

We now formally introduce the *Automated Knowledge Transfer (AutoKT)* task, including the space of algorithmic knowledge, its inherent transfer process, and an evaluation protocol.

Algorithmic knowledge. Let $\mathbb{P}$, $\mathbb{S}$, and $\mathbb{K}$ denote the problem, solver, and knowledge space. A solver $S \in \mathbb{S}$ is instantiated by its encapsulated knowledge $S(\mathcal{K})$ with $\mathcal{K} \in \mathbb{K}$. We define $S(\mathcal{K})$ as a tuple $\mathcal{K} = (\mathcal{K}_p, \mathcal{K}_a, \mathcal{K}_i)$, encompassing *problem knowledge* (e.g., constraint definition), *algorithm design knowledge* (e.g., search heuristics), and *implementation knowledge* (e.g., tensor operations and APIs).

Automated knowledge transfer (AutoKT). Given a mature source solver $S^s(\mathcal{K}^s)$ developed for a problem $P^s \in \mathbb{P}$, and a novel target problem $P^t \in \mathbb{P}$, the AutoKT task aims to leverage an LLM agent to automatically construct an effective target solver $S^t(\mathcal{K}^t)$. We formalize this process through two operators: *Local Migration* ($\mathcal{L}$) to construct the target solver skeleton from localized source knowledge, followed by *Global Migration* ($\mathcal{G}$) to coordinate the global interplay among the transferred knowledge. Both $\mathcal{L}$ and $\mathcal{G}$ rely on the source knowledge $\mathcal{K}^s$:

$$S^s(\mathcal{K}^s) \xrightarrow{\text{LLM}} S^t(\mathcal{K}^t), \quad \text{where} \quad \mathcal{K}^t = \underbrace{\mathcal{L}(\mathcal{K}^s, P^s, P^t)}_{\text{Local Migration: } \mathcal{K}^t_{\mathcal{L}}} \oplus \underbrace{\mathcal{G}(\mathcal{K}^s, P^t, \mathcal{K}^t_{\mathcal{L}})}_{\text{Global Migration: } \mathcal{K}^t_{\mathcal{G}}}. \tag{1}$$

Transfer settings. To comprehensively evaluate AutoKT, we consider three transfer settings that differ in both the relationship between the source and target problems and the availability of target-side solver knowledge: (i) *same-domain transfer*, where knowledge is transferred between variants of the same problem domain and $\mathcal{K}^s \approx \mathcal{K}^t$; (ii) *cross-domain transfer*, where source knowledge $\mathcal{K}^s$ is distilled and adapted to improve an existing target-domain solver; and (iii) *new-domain transfer*, where knowledge is transferred to a target problem from a different domain for which no existing target-domain solver is provided to the AutoKT, and $\mathcal{K}^s$ is used to construct a new solver.

4 METHODOLOGY

As illustrated in Figure 1, we propose a three-stage agentic framework for AutoKT: *Local Migration* ($\mathcal{L}$), *Global Migration* ($\mathcal{G}$), and *Validation* ($\mathcal{V}$). While $\mathcal{L}$ and $\mathcal{G}$ execute the formalized operators defined in Eq. 1, $\mathcal{V}$ acts as a simple yet necessary practical safeguard to verify the assembled solver.

4.1 LOCAL MIGRATION

Local Migration $\mathcal{L}$ constructs a target solver skeleton by adapting reusable source knowledge $\mathcal{K}^s$ to P^t. We consider two types of migration operators. For proximal domains with small semantic gaps, the ***Adaptation Migration Operator*** ($\mathcal{L}^d$) rewrites the source solver in place while preserving most of its original knowledge. For more general transfers, where direct source-to-target mapping is often infeasible, the ***Construction Migration Operator*** ($\mathcal{L}^p$) builds the target solver by selectively transferring and adapting reusable source knowledge. Both proceed progressively rather than migrating all knowledge at once, operating on the level of *knowledge components*.

Knowledge components and migration planning. A knowledge component is a semantically self-contained unit with a distinct functional role and an identifiable implementation scope, supporting a separate transfer decision. For the construction migration operator $\mathcal{L}^p$, the LLM first compiles a technical report R_s from the source solver and, when available, its accompanying paper, organizing problem, algorithm-design, and implementation knowledge. It then produces an adaptation report R_a that marks each source component as *mapped*, *replaced*, or *dropped*. We note that these marks are only initial recommendations; the final decision is made in the subsequent plan-transfer-evaluate-reflect loop. For the adaptation migration operator $\mathcal{L}^d$, since the source-target problems are close, we directly let the LLM generate the adaptation report R_a as the migration plan.

Objective. Let $k_1, \ldots, k_H$ denote the source components identified for migration. Starting from an initial target implementation S^t_0, Local Migration sequentially generates $S^t_1, \ldots, S^t_H$, where the h-th step adapts k_h into the current solver S^t_{h-1}. Each step seeks the best acceptable realization:

$$S^t_h \in \arg \max_{S \in \mathcal{C}_{\mathcal{L}}(S^t_{h-1}, k_h)} J(S), \qquad h = 1, \ldots, H, \tag{2}$$

where $J(\cdot)$ measures solver performance on P^t, and $\mathcal{C}_{\mathcal{L}}(S^t_{h-1}, k_h)$ denotes the candidate implementations reachable by transferring and refining k_h that satisfy the acceptance criterion below.

Incremental plan-transfer-evaluate-reflect loop. In each transfer step, the LLM *reads* the integration complements k_h, *transfers* it to produce a candidate $\tilde{S}^t_h$, *evaluates* its validity and performance on a held-out subset of the training instances, and *reflects* on the feedback to determine an action:

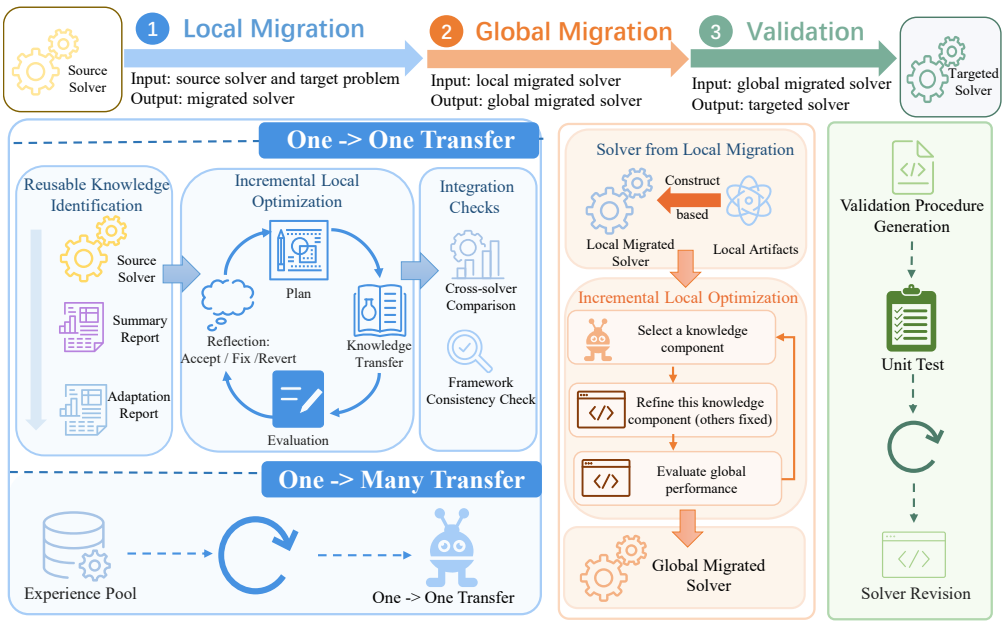


Figure 1: Framework of AutoKT. The proposed three-stage agentic framework consists of local migration, global migration, and validation. AutoKT transfers knowledge from source solvers, iteratively refines via auto-research, and validates the generated solver's feasibility.

- *Accept*: Valid realizations of *structural* components must preserve performance, $J(\tilde{S}_h^t) \geq J(S_{h-1}^t)$, while *performance-critical* components must improve it, $J(\tilde{S}_h^t) > J(S_{h-1}^t)$. The best evaluated candidate satisfying the relevant criterion is retained as S_h^t.
- *Refine*: If the criterion is not met, the LLM uses the feedback to revise k_h's implementation and its integration with the current solver, then re-evaluates the candidate.
- *Revert*: If no acceptable realization is found after three refinement attempts, set $S_h^t = S_{h-1}^t$ and retain the failure analysis for subsequent planning.

Completeness and integration checks. The resulting solver S_H^t alone does not guarantee that its implementation is complete. LLMs may silently leave placeholders, omit dependencies, or introduce logic that works locally but is inconsistent with the overall solver. We therefore maintain a component-level checklist with states $\{completed, in_progress, not_started\}$, and Local Migration terminates only after every planned component reaches a terminal state. After all components have been processed, we conduct two solver-level checks. The *cross-solver comparison* ($\mathcal{Q}_c$) compares the migrated solver against the source implementation to detect omitted functionality or knowledge. The *framework consistency check* ($\mathcal{Q}_f$) verifies that the migrated components are correctly connected and that their interfaces, execution order, and dependencies remain coherent within the target pipeline. Components that fail either check are returned to refinement before the local solver is finalized.

Experience pool for one-to-many transfer. When AutoKT is applied repeatedly within a domain, independent migrations would repeatedly encounter similar adaptation decisions and failure modes. We therefore maintain an experience pool $\mathcal{E}$ that accumulates online migration knowledge $\mathcal{K}_e$ across completed transfers. Each migration deposits a structured trajectory containing component-level decisions, successful refinements, observed failures, and their corresponding resolutions. For a new target problem, relevant experiences are retrieved to initialize migration planning and component implementation, while the newly completed trajectory is subsequently written back to the pool.

4.2 Global Migration

Local Migration focuses on the effective integration of individual knowledge components progressively and yields the artifact S_{local}^t. However, an integration chosen early in Local Migration may

become suboptimal after additional components are introduced. Global Migration addresses this remaining challenge by revisiting the transferred knowledge after assembly from a global perspective.

Objectives. Let S^t_{local} denote the solver obtained after Local Migration. Global Migration seeks to jointly refine the transferred knowledge within this solver:

$$S^t_{\text{global}} \in \arg \max_{S \in \mathcal{C}(S^t_{\text{local}})} J(S), \tag{3}$$

where $\mathcal{C}(S^t_{\text{local}})$ denotes the valid candidate implementations reachable from S^t_{local} by jointly refining the transferred components and their interactions, guided by the source knowledge $\mathcal{K}^s$. The unchanged solver is included as a fallback, and $J(\cdot)$ measures overall solver performance on P^t.

LLM-guided global coordination. We approximate this joint objective through iterative coordinate-wise refinement. At each iteration, the LLM examines the current solver and identifies a coordination opportunity among the migrated components, using the source knowledge as a reference for their intended roles and relationships. It then revises one relevant component while holding the others fixed, and evaluates the resulting solver. Beyond the component-level acceptance criterion defined above, users may also specify a global termination condition, such as reaching a target performance threshold or surpassing a baseline. The refinement process continues until this condition is met or the allocated budget is exhausted. Notably, although we let each update modify one component to structure the process, it can improve that component's interactions with others, since global Migration revisits previously accepted choices in the context of the whole assembled solver, extending the component-specific adaptations performed during Local Migration.

4.3 Validation

The migration stages primarily optimize the correctness and effectiveness of the transferred solver, but they do not explicitly ensure that the resulting solver satisfies all feasibility constraints of the target problem P^t. We introduce a validation stage $\mathcal{V}$ after Local Migration $\mathcal{L}$ and Global Migration $\mathcal{G}$ to verify the feasibility of the resulting solver $\mathcal{S}(\mathcal{K}^t_{\mathcal{G}})$ with respect to P^t.

Auto-verifying via problem-specific tests. Given a problem instance X and a candidate solution τ produced by the current solver, the LLM synthesizes an executable validator:

$$G(X, \tau) = \begin{cases} 1, & \tau \text{ satisfies all constraints,} \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

Whenever $G(X, \tau) = 0$, the reported violations are fed back to revise the solver, and the validator is re-applied until it returns 1. If the repaired solver no longer meets the user objective O^t, control is handed back to $\mathcal{R}$ for further refinement. The pipeline thus delivers the deployable target solver $S^t(\mathcal{K}^t)$, certified to satisfy both feasibility and user intent.

Final Remarks and Case Study. At each stage, the LLM is granted autonomous control over the corresponding migration process and performs the required knowledge adaptation to better fit the target problem. The benefits of this automatic design are further discussed in Appendix C.5.

5 Experiment

We evaluate AutoKT in three respects: its significant migration performance, the ablation studies of its design, and the broader insights it offers into how far LLMs can automate knowledge transfer.

Setups. AutoKT is built on Claude Opus 4.6 and its CLI agent. During solver transfer, candidates are tested on a held-out validation set disjoint from reported benchmarks, with the agent receiving only objective and feasibility feedback. The final solver is frozen and evaluated on benchmarks. Further details appear in Appendix B. Our code will be released publicly available upon acceptance.

5.1 Same-, Cross-, and New-domain Knowledge Transfer Performance

Datasets. For same-domain transfer, we use RouteFinder-style routing instances (Berto et al., 2024); for cross-domain transfer, Taillard (Taillard, 1993)/DMU (Demirkol et al., 1998) for JSSP and

Table 1: Overall performance compared to domain baselines on same-domain VRP, cross-domain JSSP/FJSP, and new-domain CPMP. VRP gaps (%) are relative to HGS-PyVRP; JSSP/FJSP gaps (%) are relative to best-known solutions. CPMP reports optimal rate (%) and average runtime (s). Gray highlights better results; "–" denotes no feasible solution or out-of-memory.

Same Domain

Solver	CVRPTW↓		CVRPL↓		CVRPLTW↓		OCVRP↓		OCVRPL↓	
	1k	5k	1k	5k	1k	5k	1k	5k	1k	5k
HGS-PyVRP	0.00 ± 0.17	0.00 ± 0.19	0.00 ± 0.12	0.00 ± 0.25	0.00 ± 0.20	0.00 ± 0.36	0.00 ± 0.29	0.00 ± 0.45	0.00 ± 0.06	0.00 ± 0.13
OR-Tools	76.18 ± 1.27	–	8.22 ± 0.04	20.20 ± 0.37	79.33 ± 0.81	–	13.98 ± 0.07	21.90 ± 0.15	13.04 ± 0.02	22.07 ± 0.09
RouteFinder	48.74	–	27.79	–	46.42	–	35.72	–	35.83	–
AFL	39.94 ± 1.89	59.56 ± 2.38	18.95 ± 1.73	32.91 ± 2.35	24.48 ± 1.83	54.53 ± 2.24	11.87 ± 0.94	21.17 ± 1.24	19.85 ± 1.13	21.67 ± 1.59
AutoKT-SISR	-2.39 ± 0.01	-0.67 ± 0.05	-0.83 ± 0.10	-0.75 ± 0.13	-2.15 ± 0.02	-0.61 ± 0.03	0.66 ± 0.01	1.06 ± 0.05	0.30 ± 0.04	2.47 ± 0.08
AutoKT-FILO	-2.12 ± 0.09	-5.96 ± 0.14	-2.38 ± 0.03	-5.93 ± 0.05	-1.55 ± 0.31	-4.97 ± 0.07	-0.77 ± 0.03	-3.67 ± 0.06	-1.01 ± 0.03	-3.72 ± 0.08

Solver	OCVRPTW↓		OCVRPLTW↓		CVRPB↓		CVRPBL↓		CVRPBTW↓	
	1k	5k	1k	5k	1k	5k	1k	5k	1k	5k
HGS-PyVRP	0.00 ± 0.24	0.00 ± 0.39	0.00 ± 0.10	0.00 ± 0.06	0.00 ± 0.18	0.00 ± 0.14	0.00 ± 0.30	0.00 ± 0.19	0.00 ± 0.19	0.00 ± 0.15
OR-Tools	58.24 ± 0.97	–	58.59 ± 0.31	–	13.49 ± 0.09	35.37 ± 0.08	13.35 ± 0.00	44.21 ± 0.16	78.39 ± 1.97	–
RouteFinder	53.22	–	54.09	–	32.16	–	38.14	–	30.58	–
AFL	21.88 ± 1.60	30.32 ± 2.18	23.41 ± 1.50	40.39 ± 2.25	27.70 ± 1.19	78.55 ± 1.67	40.97	54.05	41.01 ± 1.26	57.59 ± 1.83
AutoKT-SISR	-0.10 ± 0.08	-2.89 ± 0.09	-0.42 ± 0.06	-3.26 ± 0.05	-0.84 ± 0.06	-3.62 ± 0.09	-1.87 ± 0.06	-4.48 ± 0.04	-2.60 ± 0.06	-0.19 ± 0.05
AutoKT-FILO	-1.39 ± 0.04	-7.77 ± 0.03	-1.62 ± 0.05	-8.36 ± 0.07	-1.82 ± 0.01	-7.38 ± 0.03	-2.37 ± 0.05	-7.74 ± 0.06	-1.71 ± 0.02	-5.83 ± 0.01

Solver	CVRPBLTW↓		OCVRPB↓		OCVRPBL↓		OCVRPBTW↓		OCVRPBLTW↓	
	1k	5k	1k	5k	1k	5k	1k	5k	1k	5k
HGS-PyVRP	0.00 ± 0.22	0.00 ± 0.26	0.00 ± 0.12	0.00 ± 0.14	0.00 ± 0.17	0.00 ± 0.13	0.00 ± 0.21	0.00 ± 0.10	0.00 ± 0.18	0.00 ± 0.14
OR-Tools	76.41 ± 0.77	–	10.58 ± 0.02	22.97 ± 0.08	9.31 ± 0.21	23.04 ± 0.11	53.39 ± 0.59	–	52.39 ± 0.95	–
RouteFinder	26.13	–	29.50	–	31.84	–	35.04	–	36.11	–
AFL	28.31 ± 0.91	19.90 ± 1.58	29.89 ± 0.72	22.16 ± 0.80	25.73 ± 0.72	21.51 ± 1.01	32.73 ± 1.37	39.41 ± 1.50	33.13 ± 0.98	41.79 ± 1.29
AutoKT-SISR	-3.29 ± 0.05	-0.25 ± 0.06	0.73 ± 0.03	-3.71 ± 0.05	0.04 ± 0.07	-3.97 ± 0.05	-0.86 ± 0.09	-3.98 ± 0.05	-0.87 ± 0.07	-3.50 ± 0.07
AutoKT-FILO	-2.40 ± 0.02	-3.72 ± 0.04	-0.94 ± 0.06	-4.86 ± 0.6	-1.21 ± 0.22	-4.92 ± 0.06	-1.70 ± 0.08	-10.21 ± 0.05	-1.72 ± 0.05	-9.73 ± 0.15

Cross Domain

JSSP	SPT*	MWKR*	FDD/MWKR*	MOPNR*	L2D*	ReSched*	NS4S*	OR-Tools*	Tabu	AutoKT-JSSP
Taillard (20×15)↓	65.2	60.7	50.6	47.7	30.0	19.7	1.70	0.20	1.74	0.46 ± 0.02
Taillard (30×15)↓	61.6	52.6	45.0	45.6	33.0	21.5	1.41	2.10	1.64	0.35 ± 0.02
Taillard (30×20)↓	66.0	63.9	56.3	48.2	33.6	22.5	3.31	2.80	1.10	1.16 ± 0.02
Taillard (50×15)↓	51.4	40.9	34.8	30.1	22.4	16.1	0.00	0.00	1.93	0.00 ± 0.00
Taillard (50×20)↓	59.5	53.9	41.5	37.9	26.5	15.6	0.19	2.80	1.55	0.01 ± 0.01
DMU(30×15)↓	62.6	60.9	54.1	47.1	41.9	27.8	3.91	2.50	5.29	1.19 ± 0.02
DMU(30×20)↓	65.9	63.2	60.1	52.0	39.5	28.3	5.37	4.40	5.87	1.94 ± 0.06
DMU(40×15)↓	55.9	52.9	51.4	44.7	35.4	26.5	4.21	4.10	4.84	1.26 ± 0.13
DMU(40×20)↓	63.0	61.1	55.5	49.2	39.4	29.2	5.33	4.60	1.61	2.22 ± 0.04
DMU(50×20)↓	62.2	56.4	57.3	49.6	38.8	31.8	6.84	4.80	6.38	2.60 ± 0.09

FJSP	MWKR*	HGNN*	DANIEL*	DOAGNN*	ReSched*	2SGA*	OR-Tools	DPP-TS	AWLS	AutoKT-AWLS
Brandimarte↓	28.91	28.52	13.58	31.64	9.08	3.17	1.50	2.48	0.56	0.12 ± 0.02
Hurink↓	12.23	10.31	10.34	10.79	9.71	0.39	0.30	2.54	0.21	0.10 ± 0.02

New Domain

Metric	Osman–Christofides		Full MIP (CPLEX)		IRMA ($\alpha = 1.6$)		IRMA ($\alpha = 2.4$)		AutoKT-SISR	
	Opt.↑	Time↓	Opt.↑	Time↓	Opt.↑	Time↓	Opt.↑	Time↓	Opt.↑	Time↓
CPMP	85	180.24	75	6.83	75	3.20	100	4.71	100 ± 0	0.29

Brandimarte/Hurink for FJSP; and for new-domain transfer, OR-Library (Osman and Christofides, 1994) for CPMP and Falkenauer (Falkenauer, 1996)/BPPLib (Delorme et al., 2018) for BPP.

Domain baselines. We compare AutoKT with classical heuristics, neural methods, and LLM-based approaches across same-, cross-, and new-domain transfer. Routing: HGS-PyVRP (Wouda et al., 2024), OR-Tools (Perron and Didier), RouteFinder (Berto et al., 2024), and AFL (Zhang et al., 2025a); JSSP: L2D (Zhang et al., 2020), ReSched (Xiao et al., 2026)),NS4S (Zhang et al., 2025b), OR-Tools CP-SAT (Perron and Didier), adn Tabu (Li et al., 2025b); FJSP: some of the above baselines, and HGNN (Zhang et al., 2024), DANIEL (Park et al., 2021), DOAGNN (Zhao et al., 2025), DPP-TS (Dauzère-Pérès and Paulli, 1997), AWLS (Zhang et al., 2025c); new-domain tasks: Osman–Christofides (Osman and Christofides, 1994) and IRMA (Stefanello et al., 2015). Detailed benchmark settings and complete results are provided in Appendix C.1, C.2, and C.3.

Transfer baselines. As no established knowledge-transfer baselines exist, we evaluate Codex GPT-5.6 and Claude Opus 4.6 under two settings: *w/o source*, where the target solver is developed from scratch; and *w/ source*, where the source solver is provided as additional context.

Comparison with domain baselines. As summarized in Table 1, AutoKT demonstrates consistent gains across different transfer settings. In same-domain transfer, AutoKT-FILO outperforms HGS-PyVRP on all 30 reported 1k/5k VRP cases, achieving average gaps of -1.65% and -6.32% at the two scales, respectively. In cross-domain transfer, AutoKT-JSSP improves upon the strong tabu-search baseline, with gap reductions of up to 3.78%, while AutoKT-AWLS reduces the weighted-average FJSP gap from 0.24% to 0.10%. In new-domain CPMP, AutoKT achieves the same 100% optimality rate as the strongest baseline while reducing the average runtime from 4.71s to 0.29s. These results

Table 2: Overall performance compared to transfer baselines across same-domain, cross-domain, and new-domain transfer. VRP, JSSP, FJSP, and BPP report gaps (%, lower is better), while CPMP reports optimal rate (%, higher is better) and runtime (s, lower is better).

Method	Setting	Same-Domain		Cross-Domain				New-Domain		
		VRP		JSSP		FJSP		CPMP		BPP
		1k↓	5k↓	Taillard↓	DMU↓	Brandimarte↓	Hurink↓	Opt. Rate↑	Time (s)↓	BPPLIB↓
SOTA Reference†		0.00 ± 0.18	0.00 ± 0.21	1.17 ± 0.09	3.49 ± 0.27	0.56 ± 0.14	0.21 ± 0.10	100	4.71	0.00 ± 0.00
Codex GPT-5.6	*w/o source*	10.40 ± 1.77	15.86 ± 2.02	0.59 ± 0.17	2.56 ± 0.25	0.40 ± 0.06	0.19 ± 0.04	50	31.1	0.50 ± 0.13
	w/ source	3.03 ± 0.97	3.23 ± 0.26	0.47 ± 0.16	2.37 ± 0.32	0.26 ± 0.03	0.15 ± 0.03	65	5.5	0.28 ± 0.05
Claude Opus 4.6	*w/o source*	12.94 ± 1.91	16.99 ± 2.31	–	–	0.57 ± 0.11	0.17 ± 0.02	30	545.5	0.39 ± 0.12
	w/ source	2.82 ± 0.97	0.78 ± 0.10	0.44 ± 0.18	2.49 ± 0.35	0.20 ± 0.02	0.14 ± 0.03	65	569.6	0.12 ± 0.03
AutoKT (Claude Opus 4.6)		-0.97 ± 0.05	-1.89 ± 0.06	0.36 ± 0.02	1.54 ± 0.10	0.12 ± 0.02	0.10 ± 0.02	100 ± 0	0.29	0.02 ± 0.01

† SOTA Reference denotes the corresponding strong problem-specific baseline used for each benchmark. "−" denotes infeasible solutions.

provide evidence that transferred solver knowledge remains effective even under substantial domain shifts, yielding competitive or improved performance over strong domain-specific baselines.

Comparison with transfer baselines. We now compare AutoKT with coding agents as transfer baselines. As shown in Table 2, both agents perform poorly without the source, indicating that native LLM knowledge and reasoning capability alone are not sufficient. Providing the source improves performance, but both agents still underperform AutoKT across all three transfer settings, showing that mere source exposure is also insufficient. We further compare development costs in Table 3. AutoKT averages 22.21 minutes and 1.89M tokens, comparable to Codex with source and substantially more efficient than Claude, which is the same backbone harness but without our AutoKT designs. These results indicate that AutoKT gains from a source solver and a structured transfer pipeline rather than merely the LLM's capability.

Table 3: Development cost comparison.

Method	Setting	Same-Domain		Cross-Domain		New-Domain		Overall	
		Time	Tok.	Time	Tok.	Time	Tok.	Time	Tok.
Codex GPT-5.6	*w/o source*	28	2.2	29.00	2.10	24.50	1.40	27.74	2.11
	w/ source	23	1.6	28.50	3.00	15.50	1.00	22.79	1.68
Claude Opus 4.6	*w/o source*	92	11.1	97.50	13.20	59.00	6.95	88.94	10.87
	w/ source	63	4.3	123.50	7.45	67.50	4.80	69.84	4.68
AutoKT		10	0.7	117.00	11.00	19.00	1.75	22.21	1.89

Time is measured in minutes, and Tok. denotes LLM tokens (M). Cross-domain averages JSSP and FJSP; new-domain averages CPMP and BPP.

Results across other LLMs and agentic frameworks. We further test AutoKT across coding-agent and LLM backbones, including OpenCode + DeepSeek V3.2 and Codex + GPT-5.4; Appendix C.9 shows consistently strong performance. We also compare against stronger recent agents, including Codex + GPT Astra 6 and Claude Code + Fable 5.1 (Appendix C.10). AutoKT with a weaker backbone still outperforms them, further highlighting the value of structured knowledge transfer.

5.2 Ablation Studies and Discussions

Effects of each stage. Table 4 shows that full AutoKT performs best with 100% feasibility. Removing Validation lowers feasibility to 91.17% but leaves gaps largely unchanged, confirming its role in correctness. Removing Global Migration increases gaps across all settings, while removing both Local and Global Migration raises them further to 2.06%, 0.82%, and 0.74% for same-, cross-, and new-domain transfer. These results support the complementary roles of Local Migration for solver construction, Global Migration for knowledge coordination, and Validation for feasibility.

Effects of migration operators. We compare the *adaptation* ($\mathcal{L}^d$) and *construction* ($\mathcal{L}^p$) migration operators across increasing domain shifts. As shown in Table 4, they achieve similar same-domain gaps (0.70% vs. 0.71%), but adaptation is about $7\times$ faster through direct source reuse. Construction becomes more effective as domain distance increases, reducing the gap from 0.67% to 0.53% in cross-domain transfer and from 0.14% to 0.01% in new-domain transfer. Thus, direct adaptation suits structurally similar problems, whereas construction is more robust to larger shifts because it distills reusable knowledge and rebuilds the target solver.

Effects of global migration. To understand why Global Migration helps, we analyze pairwise interactions among transferred knowledge components before and after global coordination. For each pair (k_i, k_j), we measure their non-additive interaction as $I_{ij} = (\Delta_i + \Delta_j) - \Delta_{ij}$, where Δ_i, Δ_j, and Δ_{ij} are the increases in solution gap after removing k_i, k_j, and both, respectively. A negative I_{ij}

Table 4: Ablation studies of AutoKT. Gap is the solution gap (%; lower is better), Token is LLM usage in millions (M), and "−" denotes no feasible solution.

Setting	Same-Domain				Cross-Domain				New-Domain				Feasibility (%)
	Adaptation ($\mathcal{L}^d$)		Construction ($\mathcal{L}^p$)		Adaptation ($\mathcal{L}^d$)		Construction ($\mathcal{L}^p$)		Adaptation ($\mathcal{L}^d$)		Construction ($\mathcal{L}^p$)		
	Gap (%)	Token (M)	Gap (%)	Token (M)	Gap (%)	Token (M)	Gap (%)	Token (M)	Gap (%)	Token (M)	Gap (%)	Token (M)	
Full AutoKT	**0.70**	0.7	**0.71**	5.2	**0.67**	15.9	**0.53**	10.2	**0.14**	10.8	**0.01**	1.7	**100.00**
w/o Validation	–	0.6	–	5.0	0.67	13.6	0.53	10.0	0.14	10.1	0.01	1.2	91.17
w/o Global	0.83	0.5	0.94	4.7	0.69	12.9	0.56	9.8	0.15	8.5	0.10	1.2	100.00
w/o Global & Local	2.06	4.3	2.06	4.3	0.82	7.5	0.82	7.5	0.74	4.8	0.74	4.8	100.00

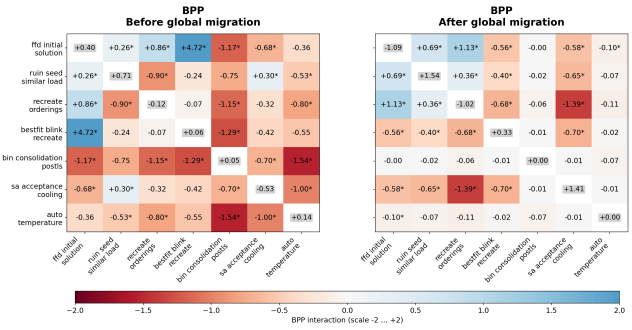


Figure 2: Pairwise knowledge interactions before and after Global Migration on BPP. Red indicates conflicts.

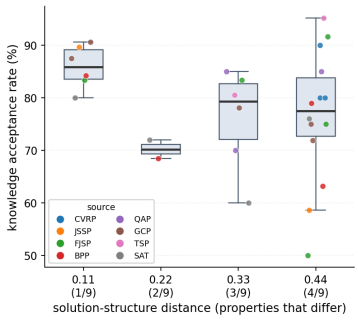


Figure 3: Knowledge acceptance rate versus solution-structure distance.

indicates redundancy. In contrast, a positive I_{ij} indicates complementarity, where the two components contribute more effectively when used together. As shown in Figure 2, the negative-interaction regions (red) shrink substantially after Global Migration, indicating that it reduces redundant or competing functionality among transferred components and improves their overall coordination. A detailed analysis of the resolved and remaining interactions is provided in Appendix C.4.

5.3 In-depth Analyses and Takeaways

How exactly does AutoKT transfer a solver? As one example, we illustrate how an SISR-based CVRP solver is transferred to the previously unseen CPMP domain. As shown in Figure 4, Local Migration progressively reduces the gap from 36.75% to 0.56%, while rejected modifications leave the solver unchanged. Global Migration then further refines interacting components, reducing the gap to 0.078%. Moreover, Appendix C.8 provides additional examples and pseudocode for representative final solvers across different transfer settings.

Which knowledge is transferable? Table 5 reports acceptance rates by transfer tier and knowledge type. Problem knowledge ($\mathcal{K}_p$), such as constraint definitions, remains domain-specific. In contrast, algorithm-design knowledge ($\mathcal{K}_a$) becomes increasingly transferable as the domain gap widens ($66.7\% \rightarrow 92.3\% \rightarrow 100\%$), because high-level principles, including local search, exploration-exploitation trade-offs, and neighborhood design, generalize across problems. Implementation knowledge ($\mathcal{K}_i$) lies between these extremes (58-74%): tensor operations and data structures are partly reusable but require "rewiring" for new environments. Transferred solvers also retain their sources' strengths: AutoKT-SISR and AutoKT-FILO preserve large-scale efficiency and outperform HGS-PyVRP at $n = 5000$ on most VRP variants (Table 9).

Table 5: Acceptance rates of knowledge types.

Source	Same	Cross	New
Problem Knowledge (K_p)	**100.00%**	0.00%	0.00%
Algorithm Design Knowledge (K_a)	66.67%	**92.31%**	**100.00%**
Implementation Knowledge (K_i)	74.07%	58.33%	60.00%

What determines knowledge transferability? We find that knowledge transfer acceptance is associated with source-target solution structural similarity, where each problem is represented by a binary vector over nine solution-structure attributes and similarity is measured by the normalized Hamming distance between vectors (Appendix C.6). Figure 3 shows that the most structurally similar pairs achieve an average acceptance rate of 85.9%, dropping as distance increases. Thus, similar solution structures facilitate knowledge reuse, while larger structural gaps require greater adaptation.

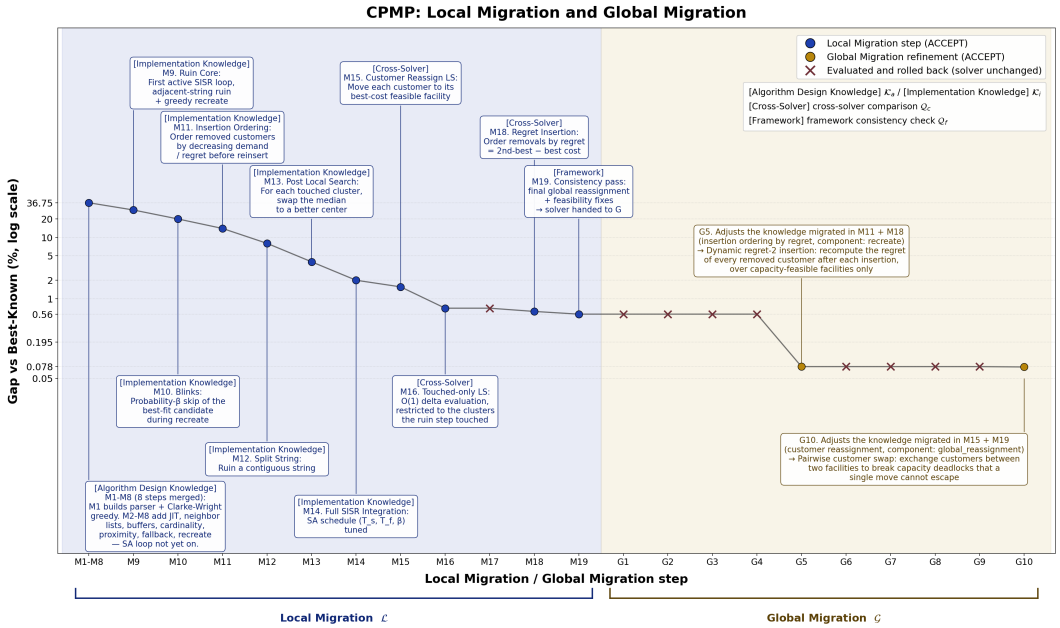


Figure 4: Overview of the local migration and global migration process. The x-axis shows the step sequence, where local migration (M1–M19) initializes the algorithm and global migration (G1–G10) iteratively refines it. The y-axis reports the gap to the best-known solution. Blue circles denote accepted local migration steps, and yellow circles denote accepted global migration steps.

AutoKT facilitates rapid baseline construction. Historically, establishing a strong baseline for a new CO variant has been a major research bottleneck. Modern neural multi-task models (e.g., MTL-POMO, MVMoE, and RouteFinder) require retraining on 10^7–10^8 instances over several GPU days, while reimplementing handcrafted heuristics demands substantial engineering effort.

In our same-domain experiments, AutoKT-FILO constructs a high-quality solver in 3–15 minutes using only five development instances and reaches a 0.20% gap to HGS-PyVRP, compared with the 6.5%–35.1% gaps of the evaluated neural baselines (Table 6). These results suggest that AutoKT can substantially reduce the effort required to establish competitive reference solvers, without replacing problem-specific benchmarking or expert solver development.

Table 6: Adaptation cost and performance on new VRP variants.

Method	Dev.	Inst.	Development Time	New var.	Gap
MTL-POMO (Liu et al., 2024b)	A100	10M	5d	retrain	20.08%
MVMoE (Zhou et al., 2024)	A100	16M	5–7d	retrain	35.10%
RouteFinder (Berto et al., 2024)	A100	2M	9–48h	retrain	6.50%
AutoKT-FILO	CPU + LLM inference	**5**	**3–15m**	LLM mig.	**0.20%**

6 Conclusion

In this paper, we formalize Automated Knowledge Transfer (AutoKT) and propose a three-stage agentic framework that systematically transfers algorithmic knowledge across combinatorial optimization problems. By leveraging existing high-quality algorithms' knowledge, AutoKT enables same-domain, cross-domain and new-domain transfer. Extensive experiments across routing, scheduling, location, and packing problems demonstrate that AutoKT produces competitive solvers across the evaluated settings. And the in-depth analysis of AutoKT offers some takeaways for the community. Despite its effectiveness, AutoKT has several limitations. First, the framework relies on the availability of high-quality source algorithms, as meaningful knowledge transfer depends on the strength of the original solver. Second, in a new-domain scenario, the success of migration depends on the compatibility between the source algorithm and the target problem. In the future, we will extend AutoKT to a broader range of problem domains and different research communities.

AI USE STATEMENT

Large language models (LLMs) were used as a writing aid to improve clarity, grammar, and overall readability of the manuscript. Also, Our proposed framework uses LLM as the backbone model to do the knowledge transfer. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Abdelkader Sbihi and Richard W Eglese. Combinatorial optimization and green logistics. *Annals of Operations Research*, 175(1):159–175, 2010.

Zhao-Yu Song, Prem Chhetri, Guanqiong Ye, and Paul Tae-Woo Lee. Green maritime logistics coalition by green shipping corridors: a new paradigm for the decarbonisation of the maritime industry. *International Journal of Logistics Research and Applications*, 28(4):363–379, 2025.

Xuexuan Zhao, Zhaokui Wang, and Gangtie Zheng. Two-phase neural combinatorial optimization with reinforcement learning for agile satellite scheduling. *Journal of Aerospace Information Systems*, 17(7):346–357, 2020.

Aymen Gannouni, Vladimir Samsonov, Mohamed Behery, Tobias Meisen, and Gerhard Lakemeyer. Neural combinatorial optimization for production scheduling with sequence-dependent setup waste. In *2020 IEEE international conference on systems, man, and cybernetics (SMC)*, pages 2640–2647. IEEE, 2020.

Ghassan Shobaki, Maxim Shawabkeh, and Najm Eldeen Abu Rmaileh. Preallocation instruction scheduling with register pressure minimization using a combinatorial optimization approach. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(3):1–31, 2013.

Petr Skobelev. Multi-agent systems for real time resource allocation, scheduling, optimization and controlling: Industrial applications. In *International Conference on Industrial Applications of Holonic and Multi-Agent Systems*, pages 1–14. Springer, 2011.

Pang Nansheng and Meng Qichen. Resource allocation in robust scheduling. *Journal of the Operational Research Society*, 74(1):125–142, 2023.

Andreas C Nearchou and Sotiris L Omirou. Differential evolution for sequencing and scheduling optimization. *Journal of Heuristics*, 12(6):395–411, 2006.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. Reevo: Large language models as hyper-heuristics with reflective evolution. *Advances in neural information processing systems*, 37:43571–43608, 2024.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. *arXiv preprint arXiv:2401.02051*, 2024a.

Xia Jiang, Yaoxin Wu, Chenhao Zhang, and Yingqian Zhang. Droc: Elevating large language models for complex vehicle routing via decomposed retrieval of constraints. In *13th International Conference on Learning Representations, ICLR 2025*. OpenReview. net, 2025.

Ni Zhang, Zhiguang Cao, Jianan Zhou, Cong Zhang, and Yew-Soon Ong. An agentic framework with llms for solving complex vehicle routing problems. *arXiv preprint arXiv:2510.16701*, 2025a.

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

Kai Li, Fei Liu, Zhenkun Wang, Xialiang Tong, Xiongwei Han, Mingxuan Yuan, and Qingfu Zhang. Ars: Automatic routing solver with large language models. *arXiv preprint arXiv:2502.15359*, 2025a.

Niels A Wouda, Leon Lan, and Wouter Kool. Pyvrp: A high-performance vrp solver package. *INFORMS Journal on Computing*, 36(4):943–955, 2024.

Jan Christiaens and Greet Vanden Berghe. Slack induction by string removals for vehicle routing problems. *Transportation Science*, 54(2):417–433, 2020.

Luca Accorsi and Daniele Vigo. A fast and scalable heuristic for the solution of large-scale capacitated vehicle routing problems. *Transportation Science*, 55(4):832–856, 2021.

Mingjie Li, Jin-Kao Hao, and Qinghua Wu. Combining hash-based tabu search and frequent pattern mining for job-shop scheduling. *IISE Transactions*, 58(6):722–737, 2025b. doi: 10.1080/24725854.2025.2532071.

Federico Berto, Chuanbo Hua, Nayeli Gast Zepeda, André Hottung, Niels Wouda, Leon Lan, Junyoung Park, Kevin Tierney, and Jinkyoo Park. Routefinder: Towards foundation models for vehicle routing problems. *arXiv preprint arXiv:2406.15007*, 2024.

Jianan Zhou, Zhiguang Cao, Yaoxin Wu, Wen Song, Yining Ma, Jie Zhang, and Chi Xu. Mvmoe: Multi-task vehicle routing solver with mixture-of-experts. *arXiv preprint arXiv:2405.01029*, 2024.

Fei Liu, Xi Lin, Zhenkun Wang, Qingfu Zhang, Tong Xialiang, and Mingxuan Yuan. Multi-task learning for routing problem with cross-problem zero-shot generalization. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 1898–1908, 2024b.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Eric Taillard. Benchmarks for basic scheduling problems. *european journal of operational research*, 64(2):278–285, 1993.

Ebru Demirkol, Sanjay Mehta, and Reha Uzsoy. Benchmarks for shop scheduling problems. *European Journal of Operational Research*, 109(1):137–141, 1998.

Ibrahim H Osman and Nicos Christofides. Capacitated clustering problems by hybrid simulated annealing and tabu search. *International Transactions in Operational Research*, 1(3):317–336, 1994.

Emanuel Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Journal of heuristics*, 2 (1):5–30, 1996.

Maxence Delorme, Manuel Iori, and Silvano Martello. Bpplib: a library for bin packing and cutting stock problems. *Optimization Letters*, 12(2):235–250, 2018.

Laurent Perron and Frédéric Didier. Cp-sat. URL `https://developers.google.com/optimization/cp/cp_solver/`.

Cong Zhang, Wen Song, Zhiguang Cao, Jie Zhang, Puay Siew Tan, and Xu Chi. Learning to dispatch for job shop scheduling via deep reinforcement learning. *Advances in neural information processing systems*, 33:1621–1632, 2020.

Xiangjie Xiao, Cong Zhang, Wen Song, and Zhiguang Cao. Resched: Rethinking flexible job shop scheduling from a transformer-based architecture with simplified states. *arXiv preprint arXiv:2603.07020*, 2026.

Junjie Zhang, Canhui Luo, Zhouxing Su, Qingyun Zhang, Zhipeng Lü, Junwen Ding, and Yan Jin. Ns4s: neighborhood search for scheduling problems via large language models. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, pages 8687–8695, 2025b.

Cong Zhang, Zhiguang Cao, Yaoxin Wu, Wen Song, and Jing Sun. Learning topological representations with bidirectional graph attention network for solving job shop scheduling problem. *Proceedings of Machine Learning Research*, 244:4192–4208, 2024.

Junyoung Park, Jaehyeong Chun, Sang Hun Kim, Youngkook Kim, and Jinkyoo Park. Learning to schedule job-shop problems: representation and policy learning using graph neural network and reinforcement learning. *International journal of production research*, 59(11):3360–3377, 2021.

Peng Zhao, You Zhou, Di Wang, Zhiguang Cao, Yubin Xiao, Xuan Wu, Yuanshu Li, Hongjia Liu, Wei Du, Yuan Jiang, et al. Dual operation aggregation graph neural networks for solving flexible job-shop scheduling problem with reinforcement learning. In *Proceedings of the ACM on Web Conference 2025*, pages 4089–4100, 2025.

Stéphane Dauzère-Pérès and Jan Paulli. An integrated approach for modeling and solving the general multiprocessor job-shop scheduling problem using tabu search. *Annals of Operations research*, 70 (0):281–306, 1997.

Junjie Zhang, Zhipeng Lü, Junwen Ding, Zhouxing Su, Xinyu Li, and Liang Gao. An effective local search algorithm for flexible job shop scheduling in intelligent manufacturing systems. *Engineering*, 50:117–127, 2025c.

Fernando Stefanello, Olinto CB de Araújo, and Felipe M Müller. Matheuristics for the capacitated p-median problem. *International Transactions in Operational Research*, 22(1):149–167, 2015.

Paolo Brandimarte. Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations research*, 41(3):157–183, 1993.

Johann Hurink, Bernd Jurisch, and Monika Thole. Tabu search for the job-shop scheduling problem with multi-purpose machines. *Operations-Research-Spektrum*, 15(4):205–215, 1994.

Haoxun Chen, Jürgen Ihlow, and Carsten Lehmann. A genetic algorithm for flexible job-shop scheduling. In *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No. 99CH36288C)*, volume 2, pages 1120–1125. Ieee, 1999.

John E Beasley. A note on solving large p-median problems. *European Journal of Operational Research*, 21(2):270–273, 1985.

Automating Combinatorial Optimization Knowledge Transfer with LLMs (Appendix)

A Problem Formulation **13**

A.1 Vehicle Routing Problems 13

A.2 Scheduling Problems 14

A.3 Capacitated p-Median Problem 14

A.4 p-Center Problem 14

A.5 One-Dimensional Bin Packing 14

B Hyperparameter **15**

C Supplement Experiment **15**

C.1 Same-domain Knowledge Transfer 15

C.2 Cross-domain Knowledge Transfer 16

C.3 New-domain Knowledge Transfer 17

C.4 Knowledge Interaction Analysis before and after Global Migration 17

C.5 Ablation Study of Autonomous Design 18

C.6 Solution-Structure Analysis of Knowledge Transferability 19

C.7 Solution-Structure Distance 20

C.8 Final Solver Artifacts 20

C.9 Applicability Across Diverse IDE And LLM Model 21

C.10 Comparison with Stronger Code Agents 22

D Prompt Example **23**

D.1 Same-domain Knowledge Transfer Prompts 23

D.2 Cross-domain Knowledge Transfer Prompts 29

D.3 New-domain Knowledge Transfer Prompts 35

A Problem Formulation

A.1 Vehicle Routing Problems

The Vehicle Routing Problem (VRP) is a fundamental combinatorial optimization problem that aims to determine optimal routes for a fleet of vehicles to serve a set of customers while minimizing the total travel cost. The Capacitated Vehicle Routing Problem (CVRP) is one of the most widely studied

variants, in which each vehicle has a limited capacity and each customer must be visited exactly once while satisfying capacity constraints.

In this work, we adopt CVRP as the source problem and extend it to a diverse set of 15 VRP variants that incorporate additional practical constraints, including time windows (TW), distance limits(L), open-route (O), and backhaul (B). Specifically, these variants include CVRP with Time Windows (CVRPTW), CVRP with distance limit constraints (CVRPL), open-route CVRP (OCVRP), and CVRP with backhaul (CVRPB), as well as their combinations such as CVRPLTW.

In CVRPTW, each customer is associated with a time window, and vehicles must arrive within the specified time interval, otherwise the solution becomes infeasible. In CVRPL, each route is subject to a maximum travel distance, requiring vehicles to complete their routes within a predefined limit. In OCVRP, vehicles are not required to return to the depot after serving customers, leading to open-ended routes. In CVRPB, vehicles perform both delivery and pickup operations within the same route, with an additional precedence constraint requiring all deliveries to be completed before any pickups. These additional constraints significantly increase the complexity of the problem by altering feasibility conditions and route structures, making these variants suitable benchmarks for evaluating the generalization ability of knowledge transfer.

A.2 Scheduling Problems

The Job Shop Scheduling Problem (JSSP) is a classical scheduling problem where a set of jobs, each consisting of a sequence of operations, must be processed on a set of machines. Each operation requires a specific machine for a fixed processing time, and the processing order of operations within each job must be respected. The objective is typically to minimize the makespan, i.e., the completion time of the last job.

The Flexible Job Shop Scheduling Problem (FJSP) extends JSSP by allowing each operation to be processed on one of multiple alternative machines, introducing an additional decision layer for machine assignment. This added flexibility significantly enlarges the search space and increases problem complexity.

A.3 Capacitated p-Median Problem

The Capacitated p-Median Problem (CPMP) is a facility location problem that aims to select p facilities from a set of candidate locations and assign each demand node to exactly one facility, while respecting capacity constraints on the facilities. The objective is to minimize the total assignment cost, typically defined as the sum of distances between demand nodes and their assigned facilities.

Unlike routing or scheduling problems, CPMP does not involve sequential decision-making over routes or operations, but instead focuses on assignment under capacity constraints. This structural difference makes CPMP a challenging target for cross-domain algorithm generation. In this work, we migrate algorithms originally designed for routing problems to CPMP, demonstrating the ability of our framework to generate competitive algorithms in structurally different problem domains.

A.4 p-Center Problem

The p-Center problem selects p facilities and assigns each demand point to a selected facility while minimizing the maximum assignment distance. Unlike CPMP's sum objective, p-Center uses a minimax objective, so improving average assignment cost is insufficient if the worst-served demand remains far from a facility. We use p-Center as an additional new-domain target and evaluate transfers from both AutoKT-CPMP and an independently developed JSSP tabu-search solver.

A.5 One-Dimensional Bin Packing

The one-dimensional Bin Packing Problem assigns items with scalar sizes to identical bins of fixed capacity while minimizing the number of used bins. Its packing representation and discrete objective differ substantially from routing, scheduling, and facility-location problems. We use it as a second additional target to test whether AutoKT can construct a functional heuristic under a larger structural shift.

B HYPERPARAMETER

Device. Unless otherwise specified, all experiments are conducted using Claude Code with Claude Opus 4.6. All experiments are conducted on a machine equipped with an AMD EPYC 9354 32-core CPU and an NVIDIA H100 NVL GPU.

Development time budget and runtime setting. The stage-level budgets are caps rather than consumed runtimes. Local Migration has a 4-hour cap and stops once all source components have been considered; Global Migration has a 4-hour cap and stops once the development objective is met; Validation has a 4-hour cap and stops once the independently generated checker verifies feasibility. In the "w/o global Migration" ablation, the removed 4-hour Migration budget is reallocated to global migration, yielding an 8-hour local migration cap while keeping source access through Q_c. For every reported AutoKT configuration, we conduct three independent runs from scratch. In each run, the coding agent is initialized in a fresh session and independently generates all migration artifacts and the final target solver. The five development instances are fixed across runs to control the evaluation environment, while the LLM trajectory and generated solver may differ across runs. Reported means and standard deviations are computed across the three independently generated solvers.

LLM usage. The reported development time includes the wall-clock time of the complete AutoKT process, including interactions with the coding agent. The number of LLM calls per run is reported in Table 7. Once migration is completed, the generated solver runs independently without any LLM inference.

Development/evaluation separation. Development uses five independently generated instances per problem variant, produced separately from all reported benchmarks. During local and global migration, the agent receives only objective and feasibility feedback on these development instances relative to the development baseline; it never accesses the reported benchmark instances, their best-known solutions, or any test-set gap information. After development terminates, the solver is frozen and evaluated once on held-out benchmarks: RouteFinder-style generated instances for routing, Taillard/DMU for JSSP, Brandimarte/Hurink for FJSP, OR-Library for CPMP, and Falkenauer (Falkenauer, 1996)/BPPLib (Delorme et al., 2018) for BPP.

Threshold. Different acceptance criteria are adopted for different problem domains. For VRP problems, we use PyVRP as the development baseline and set an acceptable quality gap of 1%. For scheduling problems, we use the method of Li et al. (Li et al., 2025b) as the development baseline with a strict 0% gap requirement for JSSP, while for FJSP we use a development reference with a 1% quality-gap threshold. For CPMP, we require the development solver to match the development reference objective.

Method/task	Coding agent	Backend	LLM calls	Dev. time	Eval. limit
AutoKT, CPMP	Claude Code	Opus 4.6	61	23 min	0.3 s
AutoKT, JSSP	Claude Code	Opus 4.6	78	118 min	300 s
AutoKT, FJSP	Claude Code	Opus 4.6	24	116 min	300 s
AutoKT, 15 VRPs	Claude Code	Opus 4.6	15–38	3–15 min	40/100/200/1000 s
AFL, re-run	Claude Code	Opus 4.6	30–60	17–30 min	40/100/200/1000 s

Table 7: Consolidated LLM-based experimental protocol.

C SUPPLEMENT EXPERIMENT

C.1 SAME-DOMAIN KNOWLEDGE TRANSFER

We consider two representative CVRP solvers, SISR (Christiaens and Vanden Berghe, 2020) and FILO Accorsi and Vigo (2021), and apply our framework to extend them to 15 additional VRP variants via direct migration operator. We compare the derived methods, AutoKT-SISR and AutoKT-FILO, against the state-of-the-art heuristic HGS-PyVRP (Wouda et al., 2024), the classical solver OR-Tools (Perron and Didier), the neural method RouteFinder (Berto et al., 2024), and LLM-based model AFL (Zhang et al., 2025a). Following RouteFinder, we evaluate on randomly generated instances with 200, 500, 1000, and 5000 nodes, using 128 instances for smaller sizes (200, 500) and 64 for larger ones (1000, 5000), with time limits of 40s, 100s, 200s, and 1000s, respectively.

Table 8: Experience pool composition.

Layer	Fields	Usage	Example
Semantic Layer	transformation_type semantic_changes invariants	Captures problem-level semantic differences of the migration case and is used for retrieving similar cases for migration.	[precedence] linehaul before backhaul ordering constraint must hold
Code Layer	code_patterns files_modified file_diffs	Provides reusable code modification patterns and guides patch generation.	[AbstractOperator.hpp, ...] AbstractOperator.hpp +15/-0 lines; handles_mixed_routes()
Debug Layer	bugs_encountered fix_methods	Guides debugging using prior failure cases and repair patterns.	incorrect boundary check is_order_compatible(...)

Table 9: Comparison of AutoKT-SISR and AutoKT-FILO against baseline methods on 15 VRP variants. Gaps (%) are reported relative to HGS-PyVRP. Negative gaps, indicating improvements over HGS-PyVRP, are highlighted in gray. "–" indicates no feasible solution or out of memory.

Solver	200	500	1k	5k	200	500	1k	5k	200	500	1k	5k
	CVRPTW				CVRPL				CVRPLTW			
HGS-PyVRP	0.00 ± 0.07	0.00 ± 0.08	0.00 ± 0.17	0.00 ± 0.19	0.00 ± 0.12	0.00 ± 0.14	0.00 ± 0.12	0.00 ± 0.25	0.00 ± 0.05	0.00 ± 0.06	0.00 ± 0.20	0.00 ± 0.36
OR-Tools	5.13 ± 0.01	10.53 ± 0.00	76.18 ± 1.27	–	6.75 ± 0.19	8.67 ± 0.02	8.22 ± 0.04	20.20 ± 0.37	5.75 ± 0.02	11.25 ± 0.12	79.33 ± 0.81	–
RouteFinder	5.38	15.87	48.74	–	5.48	16.30	27.79	–	5.73	14.67	46.42	–
AFL	6.05 ± 0.79	26.81 ± 1.26	39.94 ± 1.89	59.56 ± 2.38	9.45 ± 0.85	15.32 ± 1.10	18.95 ± 1.73	32.91 ± 2.35	7.50 ± 0.62	15.47 ± 1.06	24.48 ± 1.83	54.53 ± 2.24
AutoKT-SISR	0.33 ± 0.03	-0.34 ± 0.03	-2.39 ± 0.01	-0.67 ± 0.05	0.69 ± 0.02	0.27 ± 0.02	-0.83 ± 0.10	-0.75 ± 0.13	0.27 ± 0.06	-0.45 ± 0.06	-2.15 ± 0.02	-0.61 ± 0.03
AutoKT-FILO	0.43 ± 0.05	0.03 ± 0.07	-2.12 ± 0.09	-5.96 ± 0.14	0.05 ± 0.04	-0.56 ± 0.01	-2.38 ± 0.03	-5.93 ± 0.05	0.54 ± 0.04	0.06 ± 0.05	-1.55 ± 0.31	-4.97 ± 0.07
	OCVRP				OCVRPL				OCVRPTW			
HGS-PyVRP	0.00 ± 0.05	0.00 ± 0.15	0.00 ± 0.29	0.00 ± 0.45	0.00 ± 0.10	0.00 ± 0.04	0.00 ± 0.06	0.00 ± 0.13	0.00 ± 0.02	0.00 ± 0.07	0.00 ± 0.24	0.00 ± 0.39
OR-Tools	5.43 ± 0.01	10.33 ± 0.00	13.98 ± 0.07	21.90 ± 0.15	5.50 ± 0.11	10.43 ± 0.05	13.04 ± 0.02	22.07 ± 0.09	3.43 ± 0.04	9.31 ± 0.03	58.24 ± 0.97	–
RouteFinder	8.40	22.22	35.72	–	8.36	22.05	35.83	–	4.67	17.97	53.22	–
AFL	3.20 ± 0.52	7.01 ± 0.71	11.87 ± 0.94	21.17 ± 1.24	5.19 ± 0.69	12.35 ± 0.84	19.85 ± 1.13	21.67 ± 1.59	2.62 ± 0.76	12.15 ± 1.35	21.88 ± 1.60	30.32 ± 2.18
AutoKT-SISR	0.71 ± 0.05	0.81 ± 0.03	0.66 ± 0.01	1.06 ± 0.05	0.69 ± 0.07	0.73 ± 0.06	0.30 ± 0.04	2.47 ± 0.08	1.03 ± 0.08	0.86 ± 0.06	-0.10 ± 0.08	-2.89 ± 0.09
AutoKT-FILO	0.05 ± 0.12	-0.08 ± 0.05	-0.77 ± 0.03	-3.67 ± 0.06	0.02 ± 0.01	-0.26 ± 0.05	-1.01 ± 0.03	-3.72 ± 0.08	0.09 ± 0.01	-0.17 ± 0.01	-1.39 ± 0.04	-7.77 ± 0.03
	OCVRPLTW				CVRPB				CVRPBL			
HGS-PyVRP	0.00 ± 0.02	0.00 ± 0.03	0.00 ± 0.10	0.00 ± 0.06	0.00 ± 0.04	0.00 ± 0.20	0.00 ± 0.18	0.00 ± 0.14	0.00 ± 0.04	0.00 ± 0.05	0.00 ± 0.30	0.00 ± 0.19
OR-Tools	3.30 ± 0.24	9.43 ± 0.01	58.59 ± 0.31	–	6.94 ± 0.01	11.24 ± 0.00	13.49 ± 0.09	35.37 ± 0.08	8.29 ± 0.01	13.54 ± 0.12	13.35 ± 0.00	44.21 ± 0.16
RouteFinder	4.64	19.37	54.09	–	9.48	23.56	32.16	–	10.22	20.25	38.14	–
AFL	5.05 ± 0.74	14.40 ± 1.06	23.41 ± 1.50	40.39 ± 2.25	6.47 ± 0.46	15.74 ± 0.88	27.70 ± 1.19	78.55 ± 1.67	9.87	25.52	40.97	54.05
AutoKT-SISR	0.96 ± 0.04	1.17 ± 0.04	-0.42 ± 0.06	-3.26 ± 0.05	0.49 ± 0.02	0.06 ± 0.07	-0.84 ± 0.06	-3.62 ± 0.09	0.61 ± 0.10	0.31 ± 0.06	-1.87 ± 0.06	-4.48 ± 0.04
AutoKT-FILO	0.12 ± 0.01	0.13 ± 0.02	-1.62 ± 0.05	-8.36 ± 0.07	-0.15 ± 0.00	-0.76 ± 0.03	-1.82 ± 0.01	-7.38 ± 0.03	0.12 ± 0.03	-0.06 ± 0.05	-2.37 ± 0.05	-7.74 ± 0.06
	CVRPBTW				CVRPBLTW				OCVRPB			
HGS-PyVRP	0.00 ± 0.05	0.00 ± 0.16	0.00 ± 0.19	0.00 ± 0.15	0.00 ± 0.06	0.00 ± 0.24	0.00 ± 0.22	0.00 ± 0.26	0.00 ± 0.06	0.00 ± 0.06	0.00 ± 0.12	0.00 ± 0.14
OR-Tools	5.33 ± 0.00	9.47 ± 0.00	78.39 ± 1.97	–	5.58 ± 0.00	9.01 ± 0.05	76.41 ± 0.77	–	5.16 ± 0.00	9.60 ± 0.04	10.58 ± 0.02	22.97 ± 0.08
RouteFinder	4.70	10.70	30.58	–	4.98	9.86	26.13	–	8.94	20.30	29.50	–
AFL	11.15 ± 0.43	29.26 ± 0.81	41.01 ± 1.26	57.59 ± 1.83	6.26 ± 0.39	18.33 ± 0.62	28.31 ± 0.91	19.90 ± 1.58	14.36 ± 0.61	25.64 ± 0.42	29.89 ± 0.72	22.16 ± 0.80
AutoKT-SISR	0.27 ± 0.03	-0.56 ± 0.04	-2.60 ± 0.06	-0.19 ± 0.05	0.32 ± 0.07	-0.81 ± 0.06	-3.29 ± 0.05	-0.25 ± 0.06	1.44 ± 0.02	1.24 ± 0.04	0.73 ± 0.03	-3.71 ± 0.05
AutoKT-FILO	0.54 ± 0.00	-0.05 ± 0.02	-1.71 ± 0.02	-5.83 ± 0.01	0.68 ± 0.01	-0.07 ± 0.02	-2.40 ± 0.02	-3.72 ± 0.04	0.21 ± 0.35	-0.03 ± 0.06	-0.94 ± 0.06	-4.86 ± 0.6
	OCVRPBL				OCVRPBTW				OCVRPBLTW			
HGS-PyVRP	0.00 ± 0.09	0.00 ± 0.12	0.00 ± 0.17	0.00 ± 0.13	0.00 ± 0.04	0.00 ± 0.05	0.00 ± 0.21	0.00 ± 0.10	0.00 ± 0.03	0.00 ± 0.11	0.00 ± 0.18	0.00 ± 0.14
OR-Tools	4.89 ± 0.03	9.35 ± 0.12	9.31 ± 0.21	23.04 ± 0.11	3.57 ± 0.00	9.16 ± 0.01	53.39 ± 0.59	–	3.59 ± 0.00	8.20 ± 0.01	52.39 ± 0.95	–
RouteFinder	8.72	20.47	31.84	–	3.87	12.20	35.04	–	3.89	12.38	36.11	–
AFL	7.41 ± 0.29	19.47 ± 0.49	25.73 ± 0.72	21.51 ± 1.01	5.97 ± 0.86	21.13 ± 0.91	32.73 ± 1.37	39.41 ± 1.50	5.83 ± 0.46	21.56 ± 0.72	33.13 ± 0.98	41.79 ± 1.29
AutoKT-SISR	1.36 ± 0.02	1.42 ± 0.09	0.04 ± 0.07	-3.97 ± 0.05	0.67 ± 0.02	0.45 ± 0.02	-0.86 ± 0.09	-3.98 ± 0.05	0.64 ± 0.02	0.45 ± 0.06	-0.87 ± 0.07	-3.50 ± 0.07
AutoKT-FILO	-0.05 ± 0.03	0.05 ± 0.03	-1.21 ± 0.22	-4.92 ± 0.06	0.20 ± 0.04	-0.07 ± 0.03	-1.70 ± 0.08	-10.21 ± 0.05	0.15 ± 0.03	-0.18 ± 0.05	-1.72 ± 0.05	-9.73 ± 0.15

The results are shown in Table 9. Overall, AutoKT achieves strong performance across all VRP variants and problem sizes. AutoKT-FILO outperforms HGS-PyVRP in 71.7% of cases, while AutoKT-SISR improves in 46.7%. Both methods consistently outperform OR-Tools and RouteFinder, demonstrating that AutoKT can generate high-quality algorithms in same-domain settings.

C.2 Cross-domain Knowledge Transfer

We evaluate cross-domain transfer by applying construction migration operator to adapt the SISR algorithm from CVRP to enhance a state-of-the-art JSSP tabu search heuristic (Li et al., 2025b) and FJSP heuristic AWLS(Zhang et al., 2025c). For cross-domain scheduling tasks, we evaluate on JSSP (Taillard (Taillard, 1993), DMU (Demirkol et al., 1998)) and FJSP (Brandimarte (Brandimarte, 1993), Hurink (Hurink et al., 1994)) benchmarks. We compare against classical dispatching rules (SPT, MWKR, FDD/MWKR, MOPNR), neural solvers (L2D (Zhang et al., 2020), HGNN (Zhang et al., 2024), DANIEL (Park et al., 2021), DOAGNN (Zhao et al., 2025), ReSched (Xiao et al., 2026)), the LLM-based solver NS4S (Zhang et al., 2025b), and classical methods including OR-Tools

Table 10: Performance comparison on JSSP benchmarks. Gaps are reported relative to the best-known solutions (%). The better results between the original SOTA and AutoKT-JSSP are highlighted in gray. Entries marked with * are from ReSched (Xiao et al., 2026) and NS4S (Zhang et al., 2025b).

	Size	SPT*	MWKR*	FDD/MWKR*	MOPNR*	L2D*	ReSched*	NS4S*	OR-Tools*(3600s)	Tabu (Li et al., 2025b) (300s)	AutoKT-JSSP (300s)
Taillard	15×15	54.8	56.7	47.1	45.0	26.0	15.74	0.82	0.10	0.03	0.03 ± 0.00
	20×15	65.2	60.7	50.6	47.7	30.0	19.7	1.70	0.20	1.74	0.46 ± 0.02
	20×20	64.2	55.7	47.6	42.8	31.6	16.3	1.85	0.70	0.17	0.50 ± 0.07
	30×15	61.6	52.6	45.0	45.6	33.0	21.5	1.41	2.10	1.64	0.35 ± 0.02
	30×20	66.0	63.9	56.3	48.2	33.6	22.5	3.31	2.80	1.10	1.16 ± 0.02
	50×15	51.4	40.9	34.8	30.1	22.4	16.1	0.00	0.00	1.93	0.00 ± 0.00
	50×20	59.5	53.9	41.5	37.9	26.5	15.6	0.19	2.80	1.55	0.01 ± 0.01
	Weighted Avg	60.39	54.91	46.13	42.47	29.01	18.21	1.33	1.24	1.17	0.36 ± 0.02
DMU	20×15	64.1	62.1	53.6	49.2	39.0	23.7	2.57	1.80	0.56	0.64 ± 0.02
	20×20	64.6	58.2	52.5	45.2	37.7	22.2	2.96	1.90	2.37	0.81 ± 0.09
	30×15	62.6	60.9	54.1	47.1	41.9	27.8	3.91	2.50	5.29	1.19 ± 0.02
	30×20	65.9	63.2	60.1	52.0	39.5	28.3	5.37	4.40	5.87	1.94 ± 0.06
	40×15	55.9	52.9	51.4	44.7	35.4	26.5	4.21	4.10	4.84	1.26 ± 0.13
	40×20	63.0	61.1	55.5	49.2	39.4	29.2	5.33	4.60	1.61	2.22 ± 0.04
	50×15	50.38	48.94	52.55	40.78	36.2	26.3	5.23	3.80	1.03	1.63 ± 0.16
	50×20	62.2	56.4	57.3	49.6	38.8	31.8	6.84	4.80	6.38	2.60 ± 0.09
	Weighted Avg	61.09	57.97	54.63	47.22	38.49	26.98	4.55	3.49	3.49	1.54 ± 0.10

Table 11: Performance comparison on FJSP benchmarks. Gaps are reported relative to the best-known solutions (%). Entries marked with * are from ReSched (Xiao et al., 2026). AWLS is our re-implementation of the AWLS local search, and AutoKT-AWLS is AWLS enhanced with SISR by our framework.

Dataset	MWKR*	HGNN*	DANIEL*	DOAGNN*	ReSched*	2SGA*	OR-Tools (3600s)	DPP-TS (300s)	AWLS (300s)	AutoKT-AWLS (300s)
Brandimarte	28.91	28.52	13.58	31.64	9.08	3.17	1.50	2.48	0.56	0.12
Hurink (edata)	18.60	15.53	16.33	16.21	15.48	–	0.24	3.74	0.13	0.06
Hurink (rdata)	13.86	11.15	11.42	11.83	10.18	–	0.56	2.87	0.32	0.13
Hurink (vdata)	4.22	4.25	3.28	4.32	3.48	0.39	0.11	1.00	0.19	0.11
Weighted Avg	13.40	11.59	10.57	12.26	9.67	0.90	0.39	2.52	0.24	0.10

CP-SAT (Perron and Didier), 2SGA (Chen et al., 1999), and DPP-TS (Dauzère-Pérès and Paulli, 1997), as well as a strong heuristic (Li et al., 2025b).

As shown in Tables 10 and 11 (Appendix), AutoKT achieves strong performance across both problems. For JSSP, AutoKT-JSSP reduces the average gap from 1.17% to 0.36% on Taillard and from 3.49% to 1.54% on DMU. For FJSP, AutoKT-FJSP outperforms both neural and classical baselines and achieves the lowest gap in most cases. These results show that our framework enables effective cross-domain knowledge transfer, allowing strategies from one domain to enhance algorithm design in another domain.

C.3 New-domain Knowledge Transfer

Our framework can construct new solvers through new-domain transfer. We first transfer SISR to the Capacitated p-Median Problem (CPMP). On OR-Library instances (Osman and Christofides, 1994), AutoKT-SISR achieves a 100% optimality rate with an average runtime of 0.29s, matching the best baseline while being about $16\times$ faster.

We further evaluate transfers to p-Center (Beasley, 1985) and 1-D Bin Packing (Delorme et al., 2018). As shown in Table 12(b), AutoKT achieves average gaps of $0.84\% \pm 0.79\%$ and $0.41\% \pm 0.30\%$ on p-Center from CPMP and JSSP sources, respectively, and $0.41\% \pm 0.33\%$ and $0.02\% \pm 0.01\%$ on two Bin Packing benchmarks. These results show that AutoKT can construct effective heuristic solvers across multiple unseen target domains and is not restricted to routing-based source solvers.

C.4 Knowledge Interaction Analysis before and after Global Migration

Global Migration primarily reduces redundant interactions among peripheral components. Before refinement, `bin_consolidation_postls` and `auto_temperature` overlap with functionality already provided by the recreate and annealing components and exhibit negative interactions with many other components (e.g., -1.29 and -1.54). After refinement, both become effectively inactive, and all twelve corresponding interactions shrink to near zero. Global Migra-

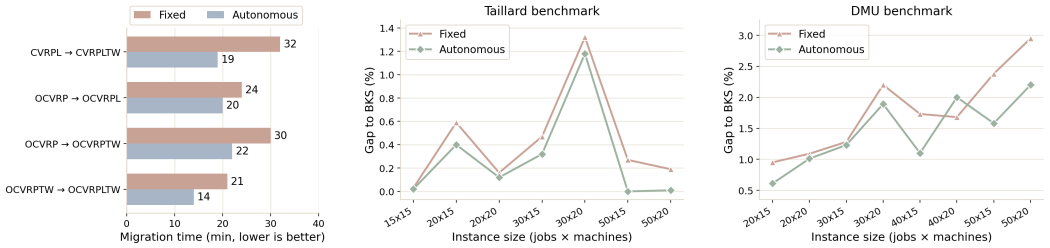
(a) Transfer to CPMP

Method	Optimal Rate (%)	Avg. Time (s)
Osman–Christofides	85	180.24
Full MIP (CPLEX)	75	6.83
IRMA ($\alpha = 1.6$)	75	3.20
IRMA ($\alpha = 2.4$)	100	4.71
AutoKT-SISR	100 ± 0	0.29

(b) Transfer to Additional New Domains

Source Solver	Target Problem	Avg. Gap
AutoKT-CPMP	p-Center (minimax)	$0.84 \pm 0.79\%$
JSSP Tabu Search	p-Center (minimax)	$0.41 \pm 0.30\%$
AutoKT-CPMP	1-D Bin Packing (Falkenauer, 1996)	$0.41 \pm 0.33\%$
AutoKT-CPMP	1-D Bin Packing (Delorme et al., 2018)	$0.02 \pm 0.01\%$

Table 12: New-domain transfer results. (a) Performance on the Capacitated p-Median Problem (CPMP), comparing AutoKT-SISR with problem-specific baselines. (b) Additional cross-domain transfer experiments, where the reference optimum for each evaluated instance is independently certified by OR-Tools CP-SAT.



(a) Same-domain migration time. (b) Taillard JSSP gap. (c) DMU JSSP gap.

Figure 5: Ablation results across different migration scenarios. The left figure shows same-domain migration time in fixed and autonomous setting, while the middle and right figures present cross-domain enhancement results on the Taillard and DMU JSSP benchmarks, respectively.

tion can also qualitatively reshape component relationships. For example, the interaction between `ruin_seed_similar_load` and `recreate_orderings` changes from -0.90 to $+0.36$, indicating a shift from substitution toward complementarity. Once the reinsertion order is made deterministic rather than randomly selected, the similar-load ruin and ordering components operate more coherently as a combined mechanism rather than as overlapping perturbations.

Global Migration does not eliminate all interactions. The remaining conflicts are mainly concentrated among core search components. In particular, SA continues to play an important role in maintaining search diversity, preserving substitutive interactions with ruin and recreate operators. Overall, Global Migration reduces redundant peripheral interactions while retaining necessary trade-offs among the core search mechanisms.

C.5 Ablation Study of Autonomous Design

We conduct an ablation study to isolate the effect of granting the LLM autonomy during knowledge transfer. Specifically, *Fixed* uses the same migration modules, tools, and feedback mechanisms as AutoKT, but follows a predefined execution pipeline, whereas *Autonomous* allows the LLM to dynamically decide which modules to invoke, how to coordinate them, and when to refine or terminate based on intermediate solver states. Both settings use Claude Code with Claude Opus 4.6 and share the same target specification, development instances, tools, feedback environment, call budget, and runtime budget, such that the comparison primarily isolates the effect of LLM autonomy.

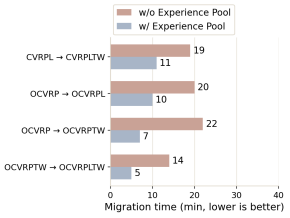


Figure 6: Effect of the experience pool on AutoKT.

As shown in Figure 5, granting the LLM autonomous control consistently improves the migration process across different transfer settings. In same-domain transfer, Autonomous substantially reduces migration time compared with the Fixed pipeline. The benefit becomes particularly pronounced in new-domain transfer (Table 13): Fixed achieves only a 75%

Table 13: Ablation study of fixed vs. autonomous design for new-domain transfer.

Metric	Fixed	Autonomous
Optimal Rate (%)	75	**100**
Avg Gap (%)	1.26	**0.00**
Avg Time (s)	0.80	**0.29**

optimality rate with a 1.26% average gap and 0.80s runtime, whereas Autonomous reaches a 100% optimality rate with a 0.00% gap and 0.29s runtime. These results indicate that AutoKT benefits not only from its migration components, but importantly from giving the LLM autonomy to adapt the transfer process to the evolving solver state rather than following a fixed, predefined workflow.

C.6 Solution-Structure Analysis of Knowledge Transferability

This appendix provides additional details on the solution-structure analysis used to study the transferability of knowledge across different combinatorial optimization problems. Our goal is to characterize how structurally similar the source and target problems are, and to examine whether this similarity is associated with the acceptance rate of migrated knowledge.

Solution-Structure representation. We characterize each problem using a binary solution-structure vector. Rather than describing surface-level problem semantics, the representation focuses on structural properties that determine how a candidate solution is organized and how its objective and constraints are evaluated.

For a problem P, we define

$$\mathbf{z}(P) = [z_{\text{seq}}, z_{\text{groups}}, z_{\text{cap}}, z_{\text{metric}}, z_{\text{adj}}, z_{\text{count}}, z_{\text{assign}}, z_{\text{prec}}, z_{\text{perm}}] \in \{0,1\}^9, \tag{5}$$

where each dimension indicates the presence or absence of a particular structural property:

- **seq**: whether the internal ordering of elements in a solution affects the objective or feasibility;
- **groups**: whether the solution partitions items into multiple containers, routes, machines, or groups;
- **cap**: whether each container, route, machine, or other resource is subject to a hard capacity constraint;
- **metric**: whether the objective is defined using distances in a metric space;
- **adj**: whether the objective depends on adjacent pairs of elements in a sequence;
- **count**: whether the objective primarily counts discrete structures, such as the number of bins or satisfied clauses, rather than accumulating a path- or distance-based cost;
- **assign**: whether each item must be assigned to a resource, location, machine, bin, color, or value;
- **prec**: whether the problem contains precedence relations among tasks or decisions;
- **perm**: whether the solution is represented as a bijective permutation over a set of items.

These attributes are intentionally coarse-grained. They are not intended to provide a complete taxonomy of combinatorial optimization problems, but rather to capture structural properties that may affect whether knowledge developed for one solver can be reused in another.

Structural profiles of the evaluated problems. Table 14 reports the structural representation used for the eight source problem families considered in our analysis.

Table 14: Binary solution-structure representations used in the structural-distance analysis.

Problem	seq	groups	cap	metric	adj	count	assign	prec	perm
CVRP	1	1	1	1	1	0	1	0	0
TSP	1	0	0	1	1	0	0	0	1
JSSP	1	1	0	0	0	0	0	1	0
FJSP	1	1	0	0	0	0	1	1	0
BPP	0	1	1	0	0	1	1	0	0
GCP	0	1	0	0	0	1	1	0	0
QAP	0	0	0	1	0	0	1	0	1
SAT	0	0	0	0	0	1	1	0	0

For example, CVRP is both sequential and grouping-based: customers are partitioned into multiple routes, while the internal ordering of customers within each route affects the travel cost. It also contains vehicle-capacity constraints and a metric, adjacency-based objective. In contrast, BPP is primarily grouping- and assignment-based: items are assigned to bins under capacity constraints, while their internal ordering does not affect the objective. TSP, meanwhile, has a strong sequential and permutation structure but does not involve grouping or capacity constraints.

These differences allow the representation to capture structural similarities that are not necessarily reflected by problem names or application domains.

C.7 Solution-Structure Distance

Given a source problem P^s and a target problem P^t, we quantify their structural discrepancy using the normalized Hamming distance between their binary structure vectors:

$$d_{\text{struct}}(P^s, P^t) = \frac{1}{9} \sum_{m=1}^{9} \mathbb{I}\left[z_m(P^s) \neq z_m(P^t)\right], \tag{6}$$

where $\mathbb{I}[\cdot]$ is the indicator function. The distance therefore lies in

$$d_{\text{struct}}(P^s, P^t) \in [0, 1], \tag{7}$$

where a smaller value indicates that the source and target problems share more solution-structure properties. For example, if two problems differ in three of the nine structural dimensions, their structural distance is

$$d_{\text{struct}} = \frac{3}{9} = 0.33. \tag{8}$$

We use the normalized distance so that the measure remains interpretable independently of the number of structural attributes.

Knowledge acceptance rate. For each source–target transfer, Local Migration identifies a set of candidate source knowledge components and determines whether each component can be successfully incorporated into the target solver.

Let $\mathcal{K}_{\text{cand}}$ denote the set of candidate knowledge components considered during migration and let $\mathcal{K}_{\text{acc}}$ denote the subset ultimately retained after adaptation, evaluation, and refinement. We define the knowledge acceptance rate as

$$A(P^s, P^t) = \frac{|\mathcal{K}_{\text{acc}}|}{|\mathcal{K}_{\text{cand}}|} \times 100\%. \tag{9}$$

This metric reflects the fraction of source knowledge that can be successfully reused after adaptation to the target problem. For every evaluated source–target pair, we associate its acceptance rate $A(P^s, P^t)$ with its corresponding structural distance $d_{\text{struct}}(P^s, P^t)$.

C.8 Final Solver Artifacts

We show compact pseudocode for every reported AutoKT solver. The provenance tags indicate which stage put each mechanism into the final solver: [L] Local Migration — source knowledge transferred as is, re-expressed for the target, or replaced by a target-native mechanism in the same slot, including the in-place rewrites of same-domain transfer; [G] Global Migration — a component introduced or re-tuned with respect to the complete assembled solver rather than in isolation; [H] host-solver knowledge retained in cross-domain transfer, where the target domain already had a solver.

Final solver pseudocode: AutoKT-SISR for CVRPLTW

```
parse CVRPLTW instance; build float distances,
  TW/service fields and the per-route distance limit                    [L]
sol ← Clarke-Wright initialization with merges that keep
  both TW and route length feasible                                     [L]
state ← SISR linked-list state + arrival times
  + cumulative route distance                                           [L]
T ← initialize SA temperature                                           [L]
```

```
for each iteration:
  cur ← copy(sol)
  ruin(cur) using SISR seed/cardinality/string/split-string logic                [L]
  recompute arrivals and route length on touched routes                          [L]
  recreate(cur) by best insertion with blink,
    filtered by TW and distance-limit feasibility                                [L][G]
  apply 2-opt on touched routes; reject reversals that
    break a time window or exceed the distance limit                             [L]
  sol ← SA_accept(cur, sol, T); update best; cool(T)                             [L]

return best
```

Final solver pseudocode: AutoKT-JSSP

```
initialize host tabu-search solver and operation-neighbor ranking                [H][L]
scale SISR ruin/recreate and SA parameters to scheduling                         [L][G]

while time remains:
  run N6 critical-block tabu search with host hash memory                        [H]
  if improved:
    snapshot current/best state                                                  [L]
  else if stagnating:
    apply host small kick                                                        [H]
    periodically run SISR-style ruin/recreate
      on machine sequences                                                       [L]
    greedily reinsert operations with precedence-cycle checks                    [L][H]
    run short tabu burst only on touched machines                                [L][H]
    accept/reject reconstructed state by SA                                      [L]

return best
```

Final solver pseudocode: AutoKT-SISR for CPMP

```
parse instance; initialize p facilities by
  Clarke-Wright greedy; assign demand to feasible facilities                     [L]
initialize neighbor lists, incremental buffers and SA schedule                   [L]

repeat within the time budget:
  cur ← copy(sol)
  SISR ruin:  adjacent-string or split-string removal
    seeded by proximity                                                          [L]
  recreate by greedy insertion with blink;
    order removed customers by demand / regret,
    dynamic regret-2 over capacity-feasible facilities                           [L][G]
  facility relocation on touched clusters
    (swap median to a better center)                                             [L]
  customer reassignment to best-cost feasible facility,
    restricted to touched clusters (O(1) delta)                                  [L]
  sol ← SA_accept(cur, sol, T)                                                   [L]
  keep best

final global reassignment + feasibility fixes,
  with pairwise customer swaps between facilities                                [L][G]

return best
```

Final solver pseudocode: AutoKT-CPMP for Bin Packing

```
parse BPP instance; sol ← first-fit-decreasing packing                           [L]
state ← bin loads, item-to-bin map, per-bin residual capacity                    [L]
T ← set SA temperature from the instance's size statistics                       [L]

for each iteration:
  cur ← copy(sol)
  ruin(cur):  pick a seed bin biased towards the lightest bins,
    then evacuate the k_s bins of most similar load                              [L][G]
  order the removed items by decreasing size when their
    mean-size-to-capacity ratio is high, shuffle otherwise                       [L][G]
  recreate(cur) by two-pass best fit:  pass 1 refuses a bin whose
    residual waste is smaller than the smallest remaining item,
    pass 2 falls back to plain best fit                                          [L][G]
  consolidate bins whose items all fit elsewhere                                 [L]
  sol ← SA_accept(cur, sol, T); update best; cool(T)                             [L]

return best
```

C.9 Applicability Across Diverse IDE and LLM Model

Table 15: Applicability across different backbones and models. Gaps are reported relative to HGS-PyVRP (%). Negative values (improvements) are highlighted in gray.

Variant	$n = 1000$			$n = 5000$		
	OpenCode+DeepSeek V3.2	Codex+GPT-5.4	Claude Code+Opus 4.6	OpenCode+DeepSeek V3.2	Codex+GPT-5.4	Claude Code+Opus 4.6
CVRPTW	-1.91	-1.83	-2.39	-2.70	-0.88	-0.67
CVRPL	-1.12	-0.66	-0.83	-0.23	-0.56	-0.75
CVRPLTW	-0.43	-1.62	-2.15	17.18	3.67	-0.61
OCVRP	0.06	2.80	0.66	5.44	12.85	1.06
OCVRPL	0.43	2.72	0.30	21.50	12.80	2.47
OCVRPTW	1.09	0.13	-0.10	9.74	-1.06	-2.89
OCVRPLTW	0.03	-0.37	-0.42	-0.75	-1.48	-3.26
CVRPB	0.99	-0.50	-0.84	-0.90	-1.21	-3.62
CVRPBL	-0.65	-1.54	-1.87	-0.72	-2.50	-4.48
CVRPBTW	-1.92	-2.02	-2.60	-0.96	-1.02	-0.19
CVRPBLTW	-3.46	-2.69	-3.29	-0.94	-0.16	-0.25
OCVRPB	-0.13	-0.08	0.73	-2.48	-1.78	-3.71
OCVRPBL	-0.08	-0.22	0.04	-0.46	-1.82	-3.97
OCVRPBTW	1.63	-0.67	-0.86	-0.35	-3.95	-3.98
OCVRPBLTW	0.92	-0.54	-0.87	-1.64	-3.40	-3.50

To evaluate the general applicability of our framework, we conduct experiments across diverse combinations of IDEs and LLMs. Specifically, we consider multiple backbone settings, including OpenCode with DeepSeek V3.2, Codex with GPT-5.4, and Claude Code with Opus 4.6.

Table 16: Generalization across different backbone models in the new-domain.

Model	Optimal Rate (%)	Avg Gap (%)	Avg Time (s)
OpenCode + DeepSeek V3.2	0	5.94	1.90
Codex + GPT-5.4	**100**	**0.00**	0.47
Claude Code + Opus 4.6	**100**	**0.00**	**0.29**

As shown in Table 15, our framework is applicable to multiple coding-agent backbones. In most cases, the generated algorithms achieve negative gaps relative to HGS-PyVRP, demonstrating that the framework can effectively transfer and optimize algorithmic strategies regardless of the underlying IDE or LLM.

Table 16 further demonstrates the performance of algorithm migration in new-domain settings across different backbone models. Overall, our framework achieves consistently strong results across all model configurations. Both GPT-based and Claude-based backbones reach perfect optimality with zero average gap, while also maintaining lower runtime. These results highlight that the effectiveness of our framework is robust across different LLMs, rather than relying on a specific model.

C.10 Comparison with Stronger Code Agents

To examine whether AutoKT's gains can be explained simply by the capability of the underlying coding model, we compare it against recent strong code agents operating without the AutoKT framework. Specifically, we evaluate Claude Code + Fable 5.1 and Codex + Astra 6 as direct-agent baselines, while AutoKT uses Claude Code + Opus 4.6 as its backbone. All methods are evaluated on the same VRP, JSSP, and CPMP tasks. In addition to downstream performance, we report development usage measured by total token consumption, providing a view of the cost required to construct the resulting solver or heuristic.

As shown in Table 17, AutoKT achieves the strongest aggregate performance across all three domains. On the 15 VRP variants, AutoKT obtains an average gap of -0.97%, compared with -0.43% for Claude Code + Fable 5.1 and $+0.02\%$ for Codex + Astra 6. The advantage extends beyond routing: AutoKT achieves lower gaps on both Taillard (0.36%) and DMU (1.54%) JSSP benchmarks, compared with $0.42\%/1.99\%$ for Fable 5.1 and $0.45\%/2.06\%$ for Astra 6, respectively. On CPMP, AutoKT reaches a 100% optimal rate, while both direct-agent baselines achieve 95%.

AutoKT also exhibits favorable development efficiency, particularly on the VRP tasks. Its average development usage is 0.7M tokens per VRP variant, substantially lower than the 4.9M tokens required by Fable 5.1 and also below the 1.2M tokens used by Astra 6.

Overall, these results suggest that increasing code-agent capability alone does not reproduce the gains obtained by AutoKT. The framework achieves better aggregate performance than both direct-agent baselines, while requiring substantially less development usage than Fable 5.1 and remaining especially efficient on the large set of VRP variants. This indicates that AutoKT's structured transfer process contributes beyond the choice of backbone model, rather than merely benefiting from a stronger coding agent.

Table 17: Comparison with stronger code agents without the AutoKT framework. VRP and JSSP results are gaps (%, lower is better), while CPMP reports the optimal rate (%, higher is better). Development usage is measured by token consumption (M).

	Agent / Model	VRP ($n = 1000$)																Other Domains		
		CVRPTW	*CVRPL*	*CVRPLTW*	*OCVRP*	*OCVRPL*	*OCVRPTW*	*OCVRPLTW*	*CVRPB*	*CVRPBL*	*CVRPBTW*	*CVRPBLTW*	*OCVRPB*	*OCVRPBL*	*OCVRPBTW*	*OCVRPBLTW*	*Avg.*	*JSSP-Tailand*	*JSSP-DMU*	*CPMP*
Performance	Claude Code Fable 5.1	-1.85	**-1.07**	-1.27	+0.66	+2.20	+0.14	-0.63	-0.47	+0.05	-2.30	-2.58	**-0.19**	+0.16	+1.44	-0.81	-0.43	0.42	1.99	95%
	Codex Astra 6	-0.19	+2.05	-1.15	+3.97	**+0.09**	**-0.50**	-0.32	-0.40	+1.04	-1.05	-2.08	-0.06	**-0.12**	-0.50	-0.43	+0.02	0.45	2.06	95%
	AutoKT + Claude Code Opus 4.6	**-2.39**	-0.83	**-2.15**	**+0.66**	+0.30	-0.10	**-0.42**	**-0.84**	**-1.87**	**-2.60**	**-3.29**	+0.73	+0.04	**-0.86**	**-0.87**	**-0.97**	**0.36**	**1.54**	**100%**
Development Usage	Claude Code Fable 5.1	10.9	7.6	0.8	5.1	4.6	4.6	3.4	5.7	6.1	0.4	7.1	5.7	4.3	4.2	2.8	4.9	22.1	22.1	2.2
	Codex Astra 6	1.1	2.2	1.0	1.2	1.1	1.0	1.1	1.0	1.4	1.3	1.2	1.3	0.9	1.1	1.4	1.2	1.7	1.7	1.2
	AutoKT + Claude Code Opus 4.6	1.0	0.6	0.4	0.7	0.4	1.9	0.4	1.1	0.5	1.0	0.5	0.6	0.6	0.9	0.5	0.7	11.8	11.8	1.4

D Prompt Example

D.1 Same-domain Knowledge Transfer Prompts

(1) Local Migration stage.

Local Migration prompt (adaptation operator $\mathcal{L}_d$)

```
You are an expert programmer migrating a solver from one problem
    variant to a closely related one. The source solver is mature:
    preserve its algorithmic knowledge and adapt it in place. Work
    autonomously --- use Read to understand code, Edit to modify files,
     and Bash to build/test.

## Task
Migrate the solver in the current directory from {source_problem} to {
    target_problem}.

## What Changes (added constraints)
{semantic_changes}

## Optimization Objective
{optimization_objective}

## Development instances and baseline
Development instances: {dev_inst_dir} (held out from every reported
    benchmark). Development baseline: {dev_baseline_file}. Candidate
    solvers are evaluated on these instances only; you receive
    objective values and feasibility, nothing else.

## Experience pool
{usage_hint}
Before planning, and before modifying each file, query the pool for
    relevant past migrations:
```

```
python query_experience.py "<filename>"

The tool returns the component-level decisions, the reference diff of
    that file, and the failures/fixes recorded by earlier transfers.
    Use them to initialise your plan and your edits. If nothing is
    returned, implement the change from the constraint semantics above;
     the absence of experience does NOT mean no change is needed.

## Step 1 --- Adaptation report (migration plan)
Because {source_problem} and {target_problem} are close, write the
    adaptation report directly, without a separate summary report.
    Decompose the source solver into knowledge components (each a self-
    contained unit with one functional role and an identifiable code
    scope: parser/state, initial solution, each ruin/recreate/local-
    search operator, acceptance/annealing, incremental evaluation, ...)
    . For every component, mark one of:
- mapped   --- reused as is or with a local change;
- replaced --- the idea carries over but the target needs a different
    mechanism in the same slot;
- dropped  --- meaningless or harmful for the target; state why.
For every added constraint, also classify its impact BEFORE writing
    code:
(a) constraint on existing moves (e.g., tighter capacity): existing
    operators may become infeasible or need updated cost evaluation;
(b) new decision variable (adds a dimension to the solution space): if
     no operator changes it inside the search loop it is frozen after
    initialisation, so ALL operators must enumerate the new dimension.
Label each component as structural (must not degrade performance) or
    performance-critical (must improve it). Save the report to {
    adaptation_report_path}. These marks are initial recommendations;
    the final decision is made in the loop below.

## Step 2 --- Component checklist
Write {plan_file}: one entry per planned component with status in {
    completed, in_progress, not_started}, refine_attempts,
    last_reflection, snapshot_dir. Migration is finished only when
    every entry is in a terminal state; never leave placeholders or to-
    dos in the code.

## Step 3 --- Plan--transfer--evaluate--reflect loop (repeat until
    Step 4)
1. Plan: re-read {plan_file} and the tail of {reflections_file}; pick
    the first component whose status is not "completed"; you may
    reorder, split, merge or drop not_started entries. Snapshot the
    source files before editing.
2. Transfer: edit ONLY the code scope of the selected component,
    following its mapped / replaced / dropped mark. Build or syntax-
    check with {build_cmd} and smoke-test with {test_cmd}.
3. Evaluate: run the solver on the development instances and compare
    against the development baseline; check feasibility of every
    written solution with the external verifier.
4. Reflect and act:
   - ACCEPT if a structural component preserves performance, or a
    performance-critical component improves it: status = completed;
    append the reflection.
   - REFINE if the criterion is not met: use the feedback to revise
    the implementation and its integration with the current solver,
    increment refine_attempts, re-evaluate.
   - REVERT if no acceptable realisation is found after three
    refinement attempts: restore the snapshot, mark the component
    completed (reverted), and record the failure analysis for later
    planning.
```

```
   Write the status back to {plan_file} before continuing.

## Step 4 --- Terminate
When every component is completed, print: LOCAL_MIGRATION_COMPLETE: <
    accepted> accepted, <reverted> reverted. The solver-level checks (
    cross-solver comparison Q_c and framework consistency check Q_f)
    and the validation stage are run by the framework afterwards.

## HARD CONSTRAINT --- NO REWRITE-FROM-SCRATCH
Preserve the existing algorithmic structure of the source solver. Add
    the target-problem constraints and any new decision variables
    incrementally on top of the existing code; do NOT rewrite the
    solver. Use Edit on pre-existing files only; do not Write a fresh
    implementation.
```

Execution example (CVRP→CVRPTW, SISR). The migration instantiated `semantic_changes` with time windows and service times, extended the parser/state with arrival times, added forward time-window checks to recreate and 2-opt, and retained the original SISR ruin and SA logic.

Solver-level checks ($\mathcal{Q}_c$ and $\mathcal{Q}_f$).

Solver-level check prompt ($\mathcal{Q}_c$, $\mathcal{Q}_f$)

```
This is a QUALITY ISSUE (the solver builds and runs but produces a
    suboptimal objective). For quality issues specifically, follow this
     analysis-first workflow before changing any code:

Step 1 --- abstract problem analysis. Without looking at code yet,
    write down STATE (what a solution encodes), ACTION (what mutates it
     per search step), and REWARD (what objective is minimised) for the
     SOURCE problem and for the TARGET problem. List how target differs
     from source on each axis.

Step 2 --- code-level comparison against the source solver at {
    source_codebase}. Using the Step-1 analysis as a checklist, check
    whether the migrated target actually reflects each expected
    difference.

Step 3 --- fix the ROOT cause identified in Step 2, not a peripheral
    symptom. Working around a structural gap with more stagnation/
    perturbation handling is not a root-cause fix.

Step 4 --- check again that the migrated components are correctly
    connected and whether their interfaces, execution order, and
    dependencies remain coherent within the complete target pipeline.

Write the Step-1 and Step-2 findings before the first build / test /
    sweep.

=== Per-instance failures ===
{per_instance_failures}
```

Execution example (SISR→CPMP). $\mathcal{Q}_c$ audits the migrated solver against the SISR source and finds four departures: the ruin step skipped facility nodes when growing a cluster, whereas the source's neighbour walk never skips a node; the annealing schedule had drifted from the source's $T_s = \text{init_cost}/N$ and $T_f = T_s/100$; post-optimisation relocated facilities but never moved an individual customer, so a customer-reassignment local search is added before the acceptance test, where the source runs its own improvement step; and that local search is restricted to the clusters touched by the ruin, as the source's 2-opt is restricted to touched routes. $\mathcal{Q}_f$ then checks how the

components connect at the end of the pipeline and adds a global reassignment pass that runs once after the search.

(2) Global Migration stage.

Component refinement prompt

```
Global Migration --- propose {m} variants of ONE knowledge component,
    all other components frozen.

Solver (template, read-only)      : {template_path}
Component under variation         : {comp_name}
  description                     : {comp_desc}
  code anchors                    : {comp_anchors}
  tunable parameters              : {comp_params}
  staged (early/late rule allowed): {comp_staged}
Current direction from the Coordinator for this component:
{direction}
Last Analyzer note for this component:
{last_note}
Known interactions / conflicts recorded so far:
{interactions}

## Task
For j = 1..{m}: copy the template to {work_dir}/{entry}.v{j}, then
    edit ONLY the code region of the component named above in that copy
     (locate it with the anchors; you may add a helper used only by
    that region). Each variant must be a genuinely different design
    choice for this component (a rule change, a different neighbourhood
    /ordering, a parameter regime, a two-stage early/late rule if
    staged), not three copies of the same tweak. Keep the CLI, the
    FINAL_OBJECTIVE line and the solution-file format unchanged. Do not
     touch any other component's code. Do not run the solver on the
    benchmark yourself; the framework evaluates the variants. A quick
    syntax/compile check is allowed ({compile_hint}).

When done, print one line per variant: VARIANT j: <one-sentence
    description of the design change>
```

Interaction analysis prompt

```
Global Migration --- causal analysis for component {comp_name}.

Template solver: {template_path}
Variants evaluated this round and their fitness on the development
    split (delta_pct < 0 means better than the template; the framework'
    s verdict is authoritative):
{variant_table}
Per-instance objectives (template vs variants):
{per_instance_table}
Component map: {components_path}

Explain, with reference to the code differences (diff the variant
    files against the template), WHY the variants performed as they did
    , and which OTHER components' behaviour the effect depends on (name
     them from the component map). Then write a JSON object to {
    note_path} with keys:
  "causal"         : <2-5 sentences>,
  "keep_direction" : <what to try next for this component>,
  "avoid"          : <what not to repeat>,
  "depends_on"     : [{"component": <name>, "kind": "supports |
    conflicts | redundant | order-sensitive", "evidence": <short>}]
```

Coordination prompt

```
Global Migration --- synergy-aware coordination across all knowledge
    components.

Template solver: {template_path}
Component map  : {components_path}
Analyzer notes : {notes_path}   (all rounds so far)
Round log      : {log_path}     (component, variants, fitness,
    accepted?)

Read everything. Then decide, for the next pass over the components:
1. A revised direction for EVERY component (one or two sentences each)
     that accounts for the interactions the Analyzers reported ---
    resolve conflicts explicitly (e.g. "do not strengthen X while Y is
    active; try weakening Y first").
2. The dependency / conflict list between components: [{"a": ..., "b":
     ..., "kind": "supports | conflicts | redundant | order-sensitive",
     "evidence": ...}].
3. The priority order of components for the next pass (most promising
    first); components that had two consecutive rounds without a
    template change may be demoted.
4. Optionally ONE joint proposal: a pair of components that should be
    varied together next round because their interaction blocks
    progress when varied alone, with a sentence on how.

Write {coord_path} as JSON with keys "directions", "dependencies", "
    priority", "joint" and "summary".
```

Execution example (CVRPLTW, SISR). Over rounds with two variants each, the only accepted refinement changed the recreate component: when reinserting a customer, routes are now scanned in order of proximity to that customer before the remaining routes, while the blink rate is kept (-0.02% on the development instances; the rival variant, a ten-fold blink rate, was worse). The final validation confirmed it under the training protocol (-0.08 pp averaged over $n = 200/500/1000$), so it became the stage output. OCVRPB shows the opposite case: three refinements accepted on the development instances, led by a "dual capacity" change that tightened the linehaul capacity, together lost 1.43 pp under the training protocol, so the final validation rejected them and the Local Migration solver was kept.

(3) Validation stage.

Verifier-generation prompt

```
Write a Python constraint-verification script for the {target_problem}
     problem.

## Target problem constraints
{constraints_text}

## File locations (absolute paths --- use exactly)
- Instance file: {inst_path}
- Solution file: {sol_path}
- Save script to: {verifier_path}

## Script requirements
1. Accepts two positional CLI args: instance_file and solution_file.
2. Parses ALL fields from the instance file (read every section).
3. Parses the solution file according to the configured format.
4. Verifies EVERY listed constraint.
5. Checks that every required entity is covered exactly once.
```

```
6. Recomputes the objective from scratch; prints "[VIOLATION] COST
    MISMATCH: reported X recomputed Y" if the reported and recomputed
    values differ by more than 1e-4.
7. Prints each violation as "[VIOLATION] <description>".
8. Prints "All constraints satisfied." and exits 0 on success.
9. Prints "TOTAL VIOLATIONS: N" and exits 1 on failure.

## Steps
1. Read the instance file to learn its exact format.
2. Read the solution file to learn its exact format.
3. Write the verifier script to: {verifier_path}.
4. Run: python "{verifier_path}" "<instance>" "<solution>".
5. Fix any Python errors and re-run until it works.
```

Execution example (CVRPTW verifier). The generated checker reparses the instance and solution, verifies customer coverage, capacity and time-window feasibility, and recomputes the objective independently from the solver output. This checker is applied only after global migration. **(3) Online knowledge summary.** Writes the structured three-layer record to the experience pool.

Online Knowledge Summary prompt

```
The {source_problem} -> {target_problem} migration just succeeded.

## What happened during the migration

### Constraint diff (semantic layer source)
{semantic_changes}

### Files modified during the migration (code layer source)
{files_modified_list}

### Debug-loop log (debug layer source)
{debug_findings}

### Experience-pool queries made during this run
{query_log_section}

## Your task --- write a structured experience-pool record

Output a single JSON object that fills every field of the three-layer
    schema. Each layer's fields are authoritative for one purpose:
    Semantic for retrieval, Code for patch generation, and Debug for
    repair.

{
  /* ---------- Semantic layer (used for retrieving similar cases) ----
     */
  "transformation_type": "<short tag derived from constraint diff, e.g.
     'distance', 'time_window', 'flexible_machine_assignment', or '_'-
    joined for compounds>",

  "semantic_changes": [
    "Added constraint: <one sentence per change, problem-agnostic>"
  ],

  "invariants": [
    "<constraint name> must be enforced in all local-search moves"
  ],

  /* ---------- Code layer (guides patch generation) ------------------
     */
  "code_patterns": [
```

```
    {
      "file_glob": "<basename of a modified file>",
      "description": "<one sentence describing the type of change this
     file received>",
      "diff_summary": "<+N/-M line summary; reusable pattern only>"
    }
  ],

  "files_modified": ["<relative path>", ...],

  "file_diffs": [
    /* full unified diffs */
  ],

  /* ---------- Debug layer (guides debugging on similar failures) ----
     */
  "bugs_encountered": [
    {
      "symptom": "<short symptom, e.g. RUNTIME CRASH, COST MISMATCH>",
      "cause": "<one-sentence root cause, problem-agnostic>",
      "fix_pattern": "<one-sentence reusable repair pattern>",
      "files": ["<files touched by this fix>"]
    }
  ],

  "fix_methods": [
    "<one-sentence reusable repair recipe per recurrent failure mode>"
  ],

  /* ---------- Notes (meta-experience: experience-pool usage) --------
     */
  "notes": {
    "query_triggers": ["<situation that triggered a query>"],
    "useful_situations": ["<when querying was valuable>"],
    "not_useful_situations": ["<when the pool returned nothing useful
    >"],
    "recommendation": "<one-line rule for future '{transformation_type
    }' migrations>"
  }
}

Rules:
- Base every entry strictly on the diff, debug log, and query log
    shown above. Do NOT invent bugs, files, or queries.
- The Semantic layer must stay problem-agnostic --- describe
    situations and patterns, never individual file names or constraint
    instances.
- The Code layer's `code_patterns` must contain reusable hints rather
    than literal diffs.
- The Debug layer must list distinct failure modes only; do not
    duplicate the same root cause across entries.
- Output valid JSON only.
```

Execution example (FILO CVRP→CVRPTW). The resulting record uses `transformation_type=time_window`, summarizes five reusable code-change patterns over 31 modified files, and records two concrete debugging failures: missing inter-route post-execute TW validation and insufficient pre-feasibility checking for 2-opt reversals.

D.2 Cross-domain Knowledge Transfer Prompts

(1) Source knowledge distillation.

Source knowledge distillation prompt

```
You are an expert algorithm designer. I will give you the full source
    code of {source_name}.

Your task is to produce a comprehensive algorithm design documentation
     that distills every significant design decision in this codebase.

GOAL OF THIS DOCUMENTATION

This document will be used by LLMs to migrate these algorithmic ideas
    to completely different combinatorial optimisation problems. The
    reader will have NO prior knowledge of {source_problem}.

HARD RULES

1. COVERAGE: every significant design decision must be documented. Do
    a final sweep of the code before finishing to check nothing is
    missed.

2. THREE-PART FORMAT: every design entry must have exactly:
   ## Summary
   ## Purpose
   ## Detail

3. ABSTRACTION RULE --- CRITICAL:
   - Summary: 1--3 sentences. Must be completely problem-agnostic. No
    mention of source-problem-specific entities (e.g. the particular
    variables, decision objects, or constraints of {source_problem}).
    Describe only the abstract algorithmic idea.
   - Purpose: explain why this design exists. State it in general
    algorithmic terms --- what class of problem it solves and what
    failure mode it prevents. Avoid problem-specific terms.
   - Detail: describe the concrete {source_problem} implementation.
    This section may be specific. Include relevant code snippets, exact
     parameter values, and known pitfalls.

4. DOCUMENT STRUCTURE: you decide the top-level organisation. Choose
    whatever structure best helps a new LLM reuse these ideas. Start
    the document with a brief explanation of the structure you chose
    and why.

5. COMPLETENESS CHECK: after writing all sections, explicitly list any
     design decisions you considered minor enough to skip, and justify
    why each one was excluded. This ensures nothing is silently dropped
    .

SOURCE CODE

{code}

{paper_section}
```

Execution example (SISR source distillation). For Sec. 1.3, the generated R_s describes simulated annealing in the required Summary–Purpose–Detail format: Boltzmann acceptance $\exp(-\Delta/T)$, geometric cooling, its exploration–exploitation purpose, and the concrete source implementation including the adaptive initial/final temperatures.

(2) Analysis of transferability via adaptation report.

Prompt of analysis of transferability via adaptation report

```
You are an expert algorithm designer. You have an existing state-of-
    the-art (SOTA) solver for {target_name} and want to improve it by
    borrowing ideas from another metaheuristic ({inspiration_name}).

## Inputs
- Existing {target_name} SOTA source: {target_sota_file}
- {inspiration_name} reference source: {inspiration_file}
- {inspiration_name} distilled design doc: {inspiration_doc_file}
- {target_name} instances: {target_inst_dir}/

## {target_name} problem
{problem_description}

## Instance file format
{instance_format}

## What to produce
Write to: {out_doc}

A {target_name} enhancement design document mirroring the structure of
     the {inspiration_name} distilled doc (same top-level headers, same
     subsection numbering). For every {inspiration_name} section,
    produce one of:

(a) mapped --- describe concretely how the idea maps onto the existing
     SOTA. Specify WHERE in the SOTA file it would be inserted (
    function / data structure / main-loop position) and WHAT {
    target_name}-native object plays the role that the source-centric
    object plays in {inspiration_name}.

(b) dropped --- name the specific property of {target_name} or of the
    existing SOTA that makes the idea redundant, harmful, or
    meaningless.

(c) replaced --- the idea carries over in spirit but must be adjusted;
     describe both the adjustment and the rationale.

Every entry MUST have:
### Summary --- abstract, problem-agnostic (1--3 sentences)
### Purpose --- why this idea matters for the SOTA
### Detail --- concrete {target_name} adaptation (code snippets,
    function/line hints in the SOTA file, data-structure changes,
    parameter hints)

## Ground rules
1. The existing SOTA is the starting point. Every applicable section
    must describe an addition or refinement, never a wholesale
    replacement of the SOTA's core.
2. Be specific. "Could borrow X" is not useful. Say WHERE in the SOTA
    file, WHICH function, and WHICH data structure.
3. Respect the SOTA's runtime characteristics. If an idea would
    increase complexity substantially, flag it in Detail so later
    phases can gate it.
4. Preserve section numbering from the inspiration doc so the planning
     phase can group sections into batches.

## Output format
Pure markdown, starting with `# {target_name} Enhancement Design (
    inspired by {inspiration_name})`. End with a completeness table
    listing every inspiration section and its mapped / dropped /
    replaced status with a one-sentence reason.
```

Execution example (SISR→CPMP adaptation). For source Sec. 4.4, R_a marks CVRP 2-opt as *replaced*: CPMP clusters are unordered, so the target analogue is facility relocation on touched clusters, which tests whether a cluster member should replace the current facility.

(3) Incremental plan-transfer-evaluate-reflect loop.

Prompt of initial plan

```
You are planning the implementation of a {target_name} solver based on
     the design document. Read the document and group its sections into
     implementation batches.

OUTPUT ONLY. Do NOT write any code. Do NOT use Edit or Write tools.

## Read this document
{target_doc}

## Task
Group the document's sections into implementation batches such that:
- Each batch is independently testable: after implementing it, the
    solver runs and produces a valid objective value.
- Batches are in dependency order.
- Each batch is small enough to implement in one focused session.

## Granularity --- MANDATORY
Each batch must correspond to exactly ONE section. The only exception
    is the very first batch, which may group the minimum sections
    needed to produce a runnable baseline that emits an objective value
     on a sample instance (typically: architecture skeleton + initial-
    solution construction + main-loop wiring with operators still
    stubbed).

## Incremental update rule --- OVERRIDES one-section-per-batch
If the design document describes an incremental cost update function,
    it MUST appear in the same batch as the first operator that relies
    on it.

## Output format --- JSON array only
[
  {
    "name": "short_name",
    "sections": ["1.1", "1.2"],
    "description": "one sentence: what this batch adds",
    "test_iterations": 0
  }
]

Fields:
- "sections": section numbers from the design doc.
- "test_iterations": 0 = no quality gate (skeleton, parameters,
    buffers, JIT compile shell, etc.); N > 0 = quality-gated; N is
    large enough to surface a real quality signal while staying under
    the per-instance wall-clock cap.

Output the JSON array only. No prose, no code.
```

Execution example (CPMP initial plan). The generated plan contains a dedicated batch `M13: post_ls` for source Sec. 4.4, separating the facility-relocation implementation from the other migration components so that it can be tested independently.

Prompt of plan-transfer-evaluate-reflect loop

```
You are the autonomous controller for a {target_name} solver migration
     based on the {inspiration_name} reference. You will run the ENTIRE
     implementation loop yourself in this single session. Do NOT stop
    after one component --- keep looping until every component in the
    plan is either "done" or "reverted".

## CONTEXT
- {target_name} design doc: {design_doc}
- Reference source: {reference_src}
- Reference distilled design doc: {reference_doc}
- Working directory: {code_dir}
- {target_name} instances: {inst_dir}
- Target entry-point file: {code_dir}/{target_entry_file}
- Plan (JSON): {plan_file}
- Reflection log: {reflections_file}
- Baseline file: {baselines_file}
- Snapshot dir: {snapshots_dir}

## Plan schema
Each component in {plan_file} is a JSON object with status in {
    completed, in_progress, not_started}, fix_attempts, last_reflection
    , and snapshot_dir.

## THE LOOP --- execute in order, repeat until TERMINATE

### Step 1 --- modify_plan
Read {plan_file} and the tail of {reflections_file}. You MAY edit the
    plan JSON: keep "completed" / "in_progress" entries; you may
    reorder, split, merge, or drop "not_started" components and insert
    new ones. If no change is needed, leave the file untouched.

### Step 2 --- pick the next component
Walk the plan in order; pick the first component whose status is not "
    completed". If every component is already accepted or reverted, go
    to TERMINATE. Resume handling: if the selected component is "
    in_progress", reuse its existing snapshot if present and continue
    from Step 4.

### Step 3 --- snapshot
Create a fresh snapshot of every source file in {code_dir} so you can
    roll back on REVERT. Store the absolute snapshot path in the
    component's snapshot_dir field and save the plan JSON.

### Step 4 --- execute (write code for the selected component)
Consult the reference {reference_src} for the section(s) you are
    implementing; translate, do not copy verbatim.

HARD SCOPE RULE:
List every function/class you intend to create or modify; for each one,
     check whether it is described in the section list of THIS
    component. Functions outside this component's section list MUST
    remain stubs.

Output-format contract:
- STDOUT: last line per instance = "FINAL_OBJECTIVE <basename> <int>".
  All other diagnostic output goes to stderr or behind a `--verbose`
    flag.
- SIDECAR: write {code_dir}/<instance_stem>.sol containing the final
    solution structure in the configured format.
- CLI: entry point accepts at least
  `<instance_path>... --seed <int> --n_iter <int>`.
```

```
Smoke-build (must succeed before declaring Step 4 complete):
{smoke_build_block}

Hard repair budget: 3 cycles total. After three failed cycles, skip
    Step 5 and emit verdict REVERT directly. Do not extend the smoke
    timeout beyond 120s.

### Step 5 --- test (incremental experiment)
Run the configured solver on the first {sample_count} instances of {
    inst_dir}/, capturing FINAL_OBJECTIVE per instance. Then run the
    external feasibility verifier on every <stem>.sol sidecar.

{self_check_block}

### Step 6 --- reflect (emit a verdict)
Single-neighbourhood mode: read {gate_verdict_file} written by the
    gate evaluator. Use its `verdict` field VERBATIM.

Operator-pool / RVND mode: follow the probe-based procedure described
    in the self-check block.

Pick exactly ONE of:
- ACCEPT --- component brings a real effect; keep it.
- REVERT --- component brings no benefit or net harm; roll back.
- REFINE --- current effect is bad but improvable on another pass.

Append one JSON line to {reflections_file}:
{"iteration": <int>, "component": "<name>",
 "verdict": "ACCEPT|REVERT|REFINE",
 "fix_attempts": <int>, "notes": "<<=300 chars>"}

### Step 7 --- act on the verdict
Re-read {plan_file}. Plan persistence is non-negotiable: Step 7 is not
     complete until you have written the new status back to {plan_file
    }.

- ACCEPT:
  status = "completed"; clear snapshot_dir; do NOT touch {
    baselines_file} (gate evaluator owns it).

- REVERT:
  remove any new file Step 4 created; restore the snapshot; set status
     = "completed"; re-run smoke test; delete the snapshot dir.

- REFINE:
  increment fix_attempts; set status = "in_progress"; keep the
    snapshot.

After the write, Read {plan_file} once and verify the status before
    proceeding.

### Step 8 --- loop
Go back to Step 1.

## TERMINATE
When every component is "completed", output exactly:

TRANSFER_V2_COMPLETE: <done count> done, <reverted count> reverted

## SAFETY GUARDRAILS
- Never modify the reference source, reference design doc, or target
    design doc.
```

```
- Re-read {plan_file} at the start of every iteration; do NOT trust
    any in-memory copy.
- Append to {reflections_file}, never overwrite.
- Do NOT use the Agent / Task / sub-agent tool at any point; every
    action must happen inline using Read / Grep / Glob / Edit / Write /
     Bash directly.
```

Execution example (CPMP plan–transfer–evaluate–reflect). Executing `M13: post_ls` implements facility relocation, runs the development gate, and returns **ACCEPT**; after this batch all five development instances reach the reference optimum (reported batch change: -33%).

Prompt of incremental evaluation (2)

```
A correct new operator can shift the firing order of other operators
    and guide the search toward a different, still valid local optimum.
     This may improve some instances while worsening others. This is
    the known trajectory effect, not necessarily a bug.

If the full sweep shows zero improvements, mixed results, or some
    regressions, run the probe procedure BEFORE rewriting the operator:

(a) Add a temporary cost_before / cost_after probe around ONLY the
    call to the new operator:

cost_before = current_cost
new_op_apply(state, ...)
cost_after = compute_cost(state, ...)
if cost_after better than cost_before:
    print("[NEW_OP] improved", file=sys.stderr)

(b) Re-run the full instance sweep with stderr captured.

(c) Inspect the result:
- If at least one "[NEW_OP] improved" line appears anywhere, the
    operator strictly improves at its call site. ACCEPT the component
    even if some final objectives regress.
- If no improvement line appears, the operator is either buggy or
    never finds an improving move. Diagnose its logic.

The moment you see any "[NEW_OP] improved" line, the component is done.
```

Execution example (JSSP incremental evaluation). For `string_positioning`, the full development sweep improves all 15/15 instances with a -1.64% change, so the component receives **ACCEPT**; the conditional call-site probe is unnecessary for this execution because the sweep already gives an unambiguous positive signal.

(4) Solver-level checks, Global Migration stage, Validation stage & Online knowledge summary. These pipeline largely reuses most of the same-domain prompt (Section D.1), since the underlying control flow all remains fundamentally unchanged across this two different transfer tiers.

D.3 New-domain Knowledge Transfer Prompts

The new-domain transfer pipeline largely reuses most of the cross-domain prompt (Section D.2), since the underlying control flow, including planning, transfer, evaluation, and reflection, all remains fundamentally unchanged across this two different transfer tiers.