

LLM-Driven Configuration of Genetic Algorithms for Constraint-Rich Optimization Problems

Harishjitu Seesadran*

hsaseendran@uri.edu

The University of Rhode Island
Kingston, RI, USA

Manbir Sodhi†

sodhi@uri.edu

The University of Rhode Island
Kingston, RI, USA

Resit Sendag‡

sendag@uri.edu

The University of Rhode Island
Kingston, RI, USA

Abstract

Configuring genetic algorithms (GAs) for new optimization problems typically requires substantial expertise in representation design, decoder construction, and constraint handling. We present a framework that combines large language models (LLMs) for problem understanding with a structured BRKGA configurator for algorithm realization, allowing users to describe optimization problems in natural language and receive executable GA implementations.

The central finding of this paper is that the value of structured LLM-assisted configuration increases with problem complexity. We compare two workflows: LLM+Config, in which the LLM extracts problem semantics and a modular BRKGA framework provides the evolutionary architecture, and LLM-Only, in which the LLM generates a complete standalone GA implementation. On standard TSP, where a simple permutation decoder is sufficient, LLM-Only performs better on large instances, showing that framework overhead can hurt when architectural support is unnecessary. In contrast, on TSP with Job Processing (TSPJ), LLM+Config achieves a 1.5% average gap from optimal and slightly outperforms the specialized state-of-the-art MGA method. On the more constrained VRP-RPD problem, the advantage of LLM+Config is decisive: the original LLM-Only workflow is on average 149.75% worse, and even after correcting its major decoder flaw, it still loses or ties in all comparisons.

These results show that direct LLM code generation may suffice for simple optimization problems, but structured configuration becomes essential when problems require multi-component encodings, simulation-based decoding, or strict constraint interaction. More broadly, the paper demonstrates that LLMs are most effective in GA design when paired with a framework that enforces valid architectural choices rather than relying on unconstrained end-to-end code generation.

CCS Concepts

• **Computing methodologies** → **Genetic algorithms**; *Neural networks*; • **Theory of computation** → **Combinatorial optimization**.

Keywords

Genetic Algorithms, Large Language Models, Algorithm Configuration, Combinatorial Optimization, Traveling Salesperson Problem, Vehicle Routing Problem

ACM Reference Format:

Harishjitu Seesadran, Manbir Sodhi, and Resit Sendag. 2026. LLM-Driven Configuration of Genetic Algorithms for Constraint-Rich Optimization Problems. In *Proceedings of Genetic and Evolutionary Computation Conference (GECCO '26)* (GECCO '26). ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 Introduction

Genetic algorithms (GAs) are widely used for combinatorial optimization, but configuring them for a new problem remains a substantial challenge. Effective performance depends not only on parameter settings, but also on architectural choices such as chromosome representation, decoder design, and constraint handling. Large language models (LLMs) offer a promising way to reduce this burden by inferring decision variables, objectives, and constraints directly from natural-language problem descriptions. However, our results show that direct end-to-end code generation is not uniformly reliable: for simple problems it may be sufficient, but for constraint-rich problems it can produce architectural mistakes that substantially degrade performance.

Algorithm configuration encompasses two distinct levels: architectural design (i.e., selecting representation schemes, decoder strategies, and constraint-handling mechanisms) and parameter tuning (i.e., optimizing population sizes, mutation rates, and selection pressure). Tools such as irace [6] and SMAC [13] address the latter, assuming a correct algorithmic architecture already exists. Our work targets the former: synthesizing decoder architectures from problem semantics. These levels are complementary; parameter tuning cannot compensate for fundamental architectural flaws, such as a decoder that prevents operation interleaving when the problem requires it.

This paper studies that distinction through two workflows. In LLM+Config, the LLM performs problem understanding while a structured BRKGA configurator determines the representation, decoder strategy, constraint-handling approach, warm starts, and implementation scaffolding. In LLM-Only, the LLM generates a complete standalone GA implementation directly. The central question is not simply whether LLMs can generate optimization code, but

*Student

†Professor

‡Professor

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GECCO '26, San Jose, Costa Rica

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

when a structured configurator adds value beyond direct generation.

Our experiments reveal a clear complexity-dependent pattern. On standard TSP, where a straightforward permutation decoder is sufficient, LLM-Only outperforms LLM+Config on larger instances, suggesting that framework overhead can hurt when architectural support is unnecessary. In contrast, on TSP with Job Processing (TSPJ), LLM+Config achieves a lower average gap from optimal than LLM-Only and slightly outperforms the specialized state-of-the-art MGA method. The difference is strongest on VRP-RPD, where LLM+Config consistently outperforms LLM-Only, and the original LLM-Only decoder is significantly worse because it fails to realize the required decoder architecture.

These findings suggest that the main value of LLM assistance in GA design is not unrestricted code generation, but the ability to map problem semantics into valid architectural choices. When the problem is simple, framework overhead may be unnecessary. When the problem is constraint-rich, however, structured configuration becomes essential for both correctness and performance.

Our contributions are threefold. First, we present a framework that separates LLM-based problem understanding from algorithm realization through a modular BRKGA configurator. Second, we develop a configurable implementation supporting multi-component chromosomes, simulation-based decoding, warm starts, and GPU-accelerated primitives such as gene-major memory layout and $O(n)$ counting sort. Third, we provide an empirical comparison across TSP, TSPJ, and VRP-RPD showing that the benefit of structured configuration scales with problem complexity, and that this distinction determines when LLM+Config should be preferred over LLM-Only.

2 Related Work

The challenge of configuring evolutionary algorithms has been studied extensively. Parameter control methods adjust parameters during search. Offline tuning approaches such as SMAC [5], irace [6], and hyperopt [14] optimize parameters of a fixed algorithmic structure. These tools are effective when the algorithm architecture—chromosome encoding, decoder logic, constraint handling—is already correctly specified. However, they cannot discover that VRP-RPD requires unified priority ordering rather than two-phase scheduling, or that TSPJ needs a multi-component chromosome. Our framework operates upstream: extracting architectural requirements from problem semantics and generating compliant decoder structures. The resulting implementation could subsequently be refined by parameter tuning tools, though we do not explore this combination here.

Recent work has explored LLMs as optimization assistants. FunSearch [3] uses LLMs to evolve mathematical functions, while EvoLLM [4] evolves heuristics for combinatorial optimization. Yang et al. [12] demonstrated that LLMs can serve as general-purpose optimizers through prompt-based search. For vehicle routing specifically, VRPAgent [11] uses LLMs to discover heuristic operators for VRP variants. Most relevant to our work, Sartori et al. [9] explore LLM-based instance-driven heuristic bias within BRKGA, though their approach focuses on heuristic selection rather than end-to-end configuration. A recent

systematic review [10] surveys the growing landscape of LLM applications in combinatorial optimization. Our work provides an end-to-end pipeline from natural language to GPU-accelerated GA implementation, with empirical comparison of LLM+Config versus LLM-only workflows.

3 Methodology

3.1 Framework Overview

The system accepts a natural language problem description and produces a configured BRKGA solver through a three-stage pipeline (Figure 1).

Stage 1: Problem Analysis. The ProblemAnalyzer class uses Claude Sonnet to parse the natural language description and extract structured problem semantics. The LLM is provided with BRKGA framework context (loaded from `context_for_llm_full.txt`) and returns a ProblemStructure object containing decision variables, objectives, constraints, and expected scale.

Stage 2: Configuration Generation. The CodeGenerator maps extracted semantics to BRKGA configuration choices. Based on problem type, it selects: chromosome structure (single vs. multi-component), decoder strategy (permutation, greedy, simulation), constraint handling (penalty, repair, multi-pass), warm start heuristics, and local search operators.

Stage 3: Code Generation. The LLM generates problem-specific C++ code that plugs into the BRKGA framework template. This includes the fitness function, decoder logic, warm start seed generation, and GPU kernel implementations. The generated code follows patterns from the framework context to ensure compatibility.

The prompts used follow a structured format. For problem analysis:

Analyze this optimization problem and extract:

1. Decision variables (name, type, count)
2. Objectives (minimize/maximize, formula)
3. Constraints (type, handling strategy)
4. Expected problem scale

Problem: {user_description}

3.2 Modular GA Configurator

The configurator produces several key components. The chromosome structure can be single or multi-component for complex encodings such as VRP-RPD’s operation priorities and agent hints. The decoder employs $O(n)$ counting sort with 256–1024 buckets for large instances. For problems with precedence constraints, the configurator generates multi-pass simulation logic. The system also produces warm start heuristics including nearest neighbor, balanced load, shortest processing time, and cluster-based approaches, along with local search operators such as 2-opt and swap on the priority component.

3.3 LLM+Config vs. LLM-Only Workflows

We compare two workflows. In the LLM+Config workflow, the LLM handles problem understanding while the BRKGA library provides the evolutionary core, GPU infrastructure, and warm starts.

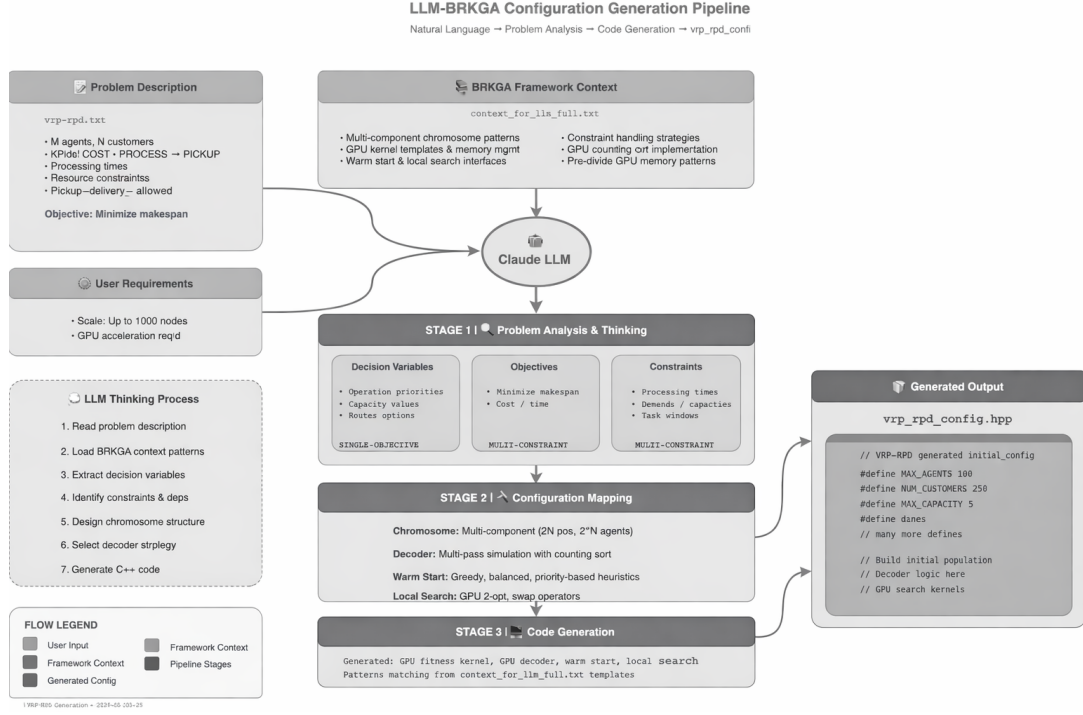


Figure 1: LLM-BRKG Configuration Generation Pipeline.

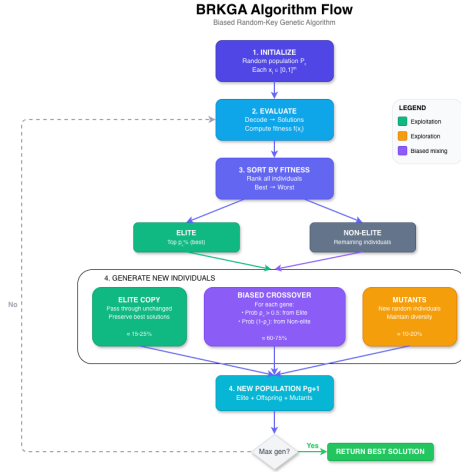


Figure 2: BRKG Algorithm Flow.

In the LLM-Only workflow, the LLM generates a complete standalone GA, which risks missing GPU acceleration, implementing incorrect constraint handling, and using inefficient algorithms.

Table 1 summarizes the VRP-RPD configuration generated by the LLM. The model determines that two chromosome components are needed for operation priorities and agent hints, selects multi-pass simulation for precedence constraints, and chooses $O(n)$ counting sort for the 2000 operations per individual.

Table 1: VRP-RPD configuration generated by LLM+Config.

Feature	Implementation
Chromosome	2 components: 2N priorities + 2N agent hints
Simulation	Multi-pass for precedence handling
Sorting	$O(n)$ counting sort, 256–1024 buckets
GPU	Per-device memory, fitness + local search kernels
Warm start	4 heuristics with perturbation

3.4 GPU-Accelerated BRKG

The framework distributes population across multiple GPUs, with each GPU maintaining independent workspace for population, fitness values, offspring, and random number generator states. Three key optimizations enable high performance. First, generator memory layout enables coalesced memory access achieving approximately 90% bandwidth utilization compared to 10–15% for individual-major layout, yielding 2–100× speedup. Second, $O(n)$ counting sort with 256–1024 buckets accelerates random-key decoding. Third, GPU-resident evolution keeps the population on GPU across generations, with host-device transfers occurring only for initialization and final retrieval.

Each CUDA thread evaluates one individual in the GPU decoder architecture. The decoder uses thread-local arrays stored in registers and local memory for decoded solutions and simulation state, avoiding shared memory contention. For VRP-RPD, the kernel maintains per-thread arrays for agent times, locations, and inventories during multi-pass simulation:

```

__global__ void fitness_kernel(T* pop, T* fitness, ...) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    T* chrom = pop + idx * chrom_len;
    int order[MAX_OPS]; // Thread-local decoded order
    T agent_time[MAX_AGENTS]; // Simulation state
    counting_sort(chrom, order, n_ops); // O(n)
    T makespan = simulate_multipass(order, ...);
    fitness[idx] = makespan;
}

```

4 Experimental Setup

4.1 Benchmark Problems

For the Traveling Salesperson Problem (TSP), we use standard TSPLIB instances [8] ranging in size from 17 nodes (gr17) to 1000 nodes (dsj1000), with Euclidean distances computed from city coordinates.

For the TSP with Job Processing (TSPJ) [1], each city has a job requiring processing time. This problem requires two chromosome components encoding the city sequence and job assignment. The objective is to minimize makespan, with job times drawn uniformly from [10, 50] for base instances or [20, 100] for 2x instances.

For the Vehicle Routing Problem with Resource Pick-up and Delivery (VRP-RPD), M agents service N customers, where each customer requires a sequence of DROP-OFF, PROCESSING, and PICK-UP operations. The problem includes precedence constraints requiring pickup to occur after dropoff plus processing time, capacity constraints, and the flexibility for cross-agent pickup. The objective is to minimize makespan. We use TSPLIB distance matrices with processing times uniform in [20, 100], testing configurations with 3–10 agents and capacity 2–5.

4.2 Baselines

For TSP, we compare against optimal solutions from TSPLIB and the Concorde exact solver [7]. For TSPJ, we compare against optimal and best known solutions from the literature [1], as well as a multioperator genetic algorithm (MGA) [2]. For VRP-RPD, no standard benchmarks exist as yet; we compare LLM+Config against LLM-Only, and also an Adaptive Large Neighborhood Search meta-heuristic [15] to evaluate framework contribution.

4.3 Implementation Details

All experiments were conducted on NVIDIA L40 GPUs with 48GB memory, using up to 4 GPUs. The LLM used was Claude Sonnet with temperature 0.1. BRKGA parameters include population size of $\min(2000 \times \text{GPUs}, 20n)$ total across all GPUs, elite fraction of 20%, mutant fraction of 10%, and generations ranging from 1000 to 10000 scaling with problem size n . Warm start uses 20 seeds, and local search is applied every 10 generations. Each configuration was run 10 times with different random seeds, and we report the median.

5 Results

5.1 Performance on TSP Variants

Table 2 compares LLM+Config and LLM-Only on standard TSP instances. Both approaches achieve optimal solutions on small instances with $n \leq 24$. For medium instances, LLM+Config shows slight advantages. However, a crossover occurs on large instances where LLM-Only outperforms LLM+Config: on gr666, LLM-Only achieves 8.84% gap versus 23.87% for LLM+Config; on rat783, 9.65% versus 24.30%; and on dsj1000, 8.61% versus 32.00%. This reversal reflects TSP’s simple constraint structure, where the framework overhead provides no benefit.

Table 2: TSP performance comparison. Gap columns show percentage above optimal; Δ = LLM-Only gap – LLM+Config gap (negative favors LLM-Only).

Instance	n	Optimal	LLM+Config Gap	LLM-Only Gap	Δ
gr17	17	2,085	0.00%	0.00%	0.00
gr21	21	2,707	0.00%	0.00%	0.00
berlin52	52	7,542	0.34%	1.84%	+1.50
kroA100	100	21,282	0.51%	2.64%	+2.13
eil101	101	629	3.50%	3.82%	+0.32
gr202	202	40,160	4.92%	4.82%	−0.10
gr431	431	171,414	6.60%	6.51%	−0.09
gr666	666	294,358	23.87%	8.84%	−15.03
rat783	783	8,806	24.30%	9.65%	−14.65
dsj1000	1000	18,659,688	32.00%	8.61%	−23.39

Table 3 presents results on TSP with Job Processing [1], comparing LLM+Config, LLM-Only, and the best local search heuristic (LSI) from Mosayebi et al. Gap columns show percentage difference from optimal for each method. We use the full benchmark set introduced in [1].

Table 3: TSPJ performance comparison. Gap columns show percentage above optimal; Δ = LLM-Only gap – LLM+Config gap.

Instance	LLM+ Config	LLM- Only	LSI	Opt.	Gap (%)		Δ
					+Config	Only	
gr17	2760	2760	2760	2760	0.0	0.0	0.0
gr21	7868	7852	7956	7788	1.0	0.8	−0.2
gr24	1812	1823	1818	1806	0.3	0.9	+0.6
fri26	1283	1283	1326	1283	0.0	0.0	0.0
bays29	2916	2916	2940	2937	−0.7	−0.7	0.0
gr48	7509	7572	7499	7288	3.0	3.9	+0.9
eil51	636	663	640	630	1.0	5.2	+4.2
berlin52	11087	11571	11226	11087	0.0	4.4	+4.4
eil76	840	847	822	802	4.7	5.6	+0.9
eil101	1003	1029	976	947	5.9	8.7	+2.8
Avg.					1.5	2.9	+1.4

LLM+Config achieves 1.5% average gap from optimal, outperforming both LLM-Only (2.9%) and the specialized MGA algorithm (1.63%) [2] without manual tuning.

Table 4 compares our LLM+Config framework against the multi-operator genetic algorithm (MGA) recently proposed by Gutiérrez-Aguirre and Contreras-Bolton [2], which represents the current best metaheuristic for TSPJ. The MGA employs five initial population procedures, three crossover operators, and six mutation operators with extensive parameter tuning via irace.

Table 4: Comparison against state-of-the-art methods on TSPLIB TSPJ instances.

Method	Avg. Makespan	Gap from BKS
CPLEX Optimal [1]	3,729.79	0.00%
LLM+Config	3,771.40	1.5%
MGA [2]	3,790.16	1.63%
LSH-III [1]	3,819.61	2.41%
LSH-IV [1]	3,838.23	2.91%

Our LLM+Config framework achieves a 1.5% average gap from optimal, outperforming the specialized MGA (1.63% gap) despite the MGA being purpose-built for TSPJ with extensive operator tuning. This demonstrates that the framework’s automatic configuration produces competitive results even against state-of-the-art problem-specific algorithms, without requiring manual algorithm design or parameter optimization.

5.2 Performance on VRP-RPD

VRP-RPD represents the most complex variant in our evaluation, requiring multi-agent coordination, precedence constraints, and resource inventory management. Although a forthcoming paper contains a broader treatment of the problem, the description is sufficient with respect to the formulation, algorithms, and experiments used here.

The framework handles these challenges through multi-pass simulation, guaranteeing feasibility by construction. GPU acceleration enables efficient handling of instances with up to 1,000 customers, achieving 20–29× speedup over LLM-Only.

5.3 Workflow Comparison

Table 5 presents a summary comparison between LLM+Config and LLM-Only across 12 TSPLIB instances, with results aggregated across 5 constraint variants per instance. The original LLM-Only decoder used two-phase scheduling, so we attempted correction by re-prompting with additional emphasis on interleaving, which was already specified in the initial prompt. Full per-variant results are provided in the supplementary material.

The LLM+Config approach outperforms the original uncorrected LLM-Only in all 61 valid comparisons, with LLM-Only producing solutions that are on average 149.75% worse in makespan. The most extreme case is gr24_5x with 278.2% degradation for LLM-Only.

After correcting the two-phase decoder flaw, LLM+Config wins 43 of 57 complete comparisons (75%), ties in 14 cases (25%), and loses in none. The correction dramatically improves small instances: for $n \leq 52$, gap reductions range from 70–276 percentage points. However, on the largest instances including gr431_base, gr431_2x, gr666_1R10, and gr666_1R20, the correction actually

Table 5: VRP-RPD workflow comparison. LLM-Only = original two-phase decoder; LLM-Only* = after interleaving correction. Gap columns show percentage above LLM+Config baseline; Δ = improvement from correction; Ties = variants matching baseline exactly.

Instance	n	Gap vs. Baseline		Correction	
		LLM-Only	LLM-Only*	Δ	Ties
gr17	17	+122.3%	+0.2%	+122.0%	3/5
gr21	21	+137.6%	+4.5%	+133.1%	3/5
gr24	24	+193.6%	+9.4%	+184.2%	3/5
bays29	29	+196.6%	+1.6%	+195.0%	3/5
gr48	48	+176.6%	+16.4%	+160.3%	1/5
eil51	51	+186.3%	+12.0%	+174.3%	1/5
berlin52	52	+196.4%	+14.0%	+182.4%	1/5
kroA100	100	+163.0%	+29.5%	+133.5%	0/5
eil101	101	+170.1%	+30.9%	+139.1%	0/5
gr202	202	+127.0%	+37.5%	+89.5%	0/5
gr431	431	+90.5%	+87.3%*	+3.2%	0/5
gr666	666	+54.3%	+84.3%*	−30.0%*	0/2

hurts performance with negative gap reduction, suggesting that naively adding interleaving without the framework’s other optimizations can degrade performance on complex instances.

Both approaches show increased makespan for constrained variants such as 1R10 and 1R20, but LLM+Config maintains consistent performance ratios across constraint types. On many constrained variants with small instances, the corrected LLM-Only achieves exact ties with 0% gap relative to LLM+Config. Section 6.2 provides a detailed analysis of the architectural differences that explain this performance gap.

6 Discussion

6.1 Debugging Episode

The LLM-Only decoder contained a critical architectural flaw despite explicit interleaving requirements in the problem specification. When confronted, the LLM initially defended its implementation, and only concrete code-level proof convinced it to acknowledge the bug. This episode illustrates that LLMs can confidently produce code that violates explicit requirements while claiming compliance.

The key insight from our evaluation is that problem complexity determines framework value. For TSP, the simple constraint structure means that framework overhead actually hurts performance. In contrast, VRP-RPD’s complex constraints require structured handling that the framework provides.

6.2 Decoder Architecture Comparison

Figure 3 illustrates the key architectural difference between the two approaches. The LLM-Only decoder processes all dropoffs before any pickups within each pass, while LLM+Config uses unified priority ordering that enables true interleaving, achieving 54% makespan reduction in the illustrated example.

As shown in Table 6, LLM+Config uses 4N genes consisting of 2N operation priorities and 2N agent hints with unified priority scope, while LLM-Only uses 2N genes with separate dropoff and

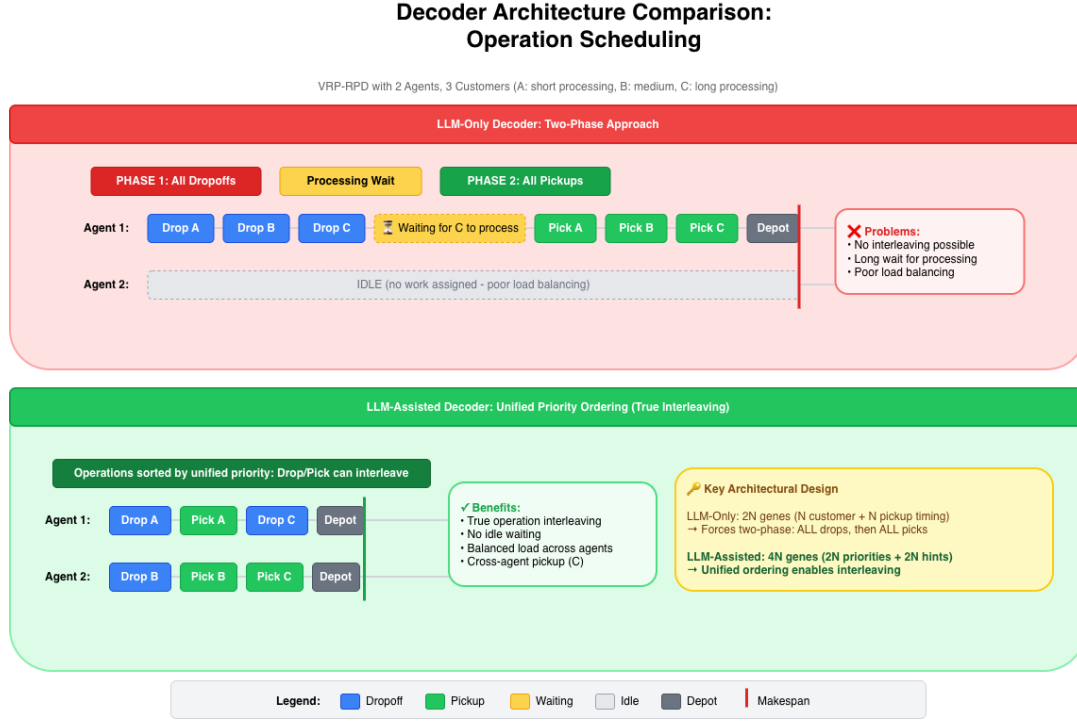


Figure 3: Decoder architecture comparison: two-phase (top) vs. unified priority ordering (bottom).

pickup phases. This architectural limitation cannot be overcome by running additional generations.

Table 6: Chromosome structure comparison for VRP-RPD with N customers.

Aspect	LLM+Config	LLM-Only
Total genes	4N	2N
Component 1	Operation priorities (2N)	Customer priorities (N)
Component 2	Agent hints (2N)	Pickup timing (N)
Priority scope	Unified (drops + picks)	Separate (drops only)
Interleaving	Yes (within each pass)	No (two-phase per pass)

Beyond the interleaving difference, LLM+Config uses chromosome-guided agent hints instead of LLM-Only’s greedy assignment, and employs $O(n)$ counting sort instead of $O(n^2)$ linear scans, contributing to 10–200× speedup.

6.3 Limitations

The current framework is limited to combinatorial optimization problems expressible as permutation, assignment, or selection. This scope covers a substantial class of routing and sequencing problems, but it does not yet address continuous, mixed, or multi-objective optimization. Configuration quality also depends on LLM capabilities, although the structured configurator helps maintain validity by constraining code generation to a fixed set of architectural choices. LLM outputs may still vary between

runs, so exact reproducibility requires fixed random seeds, fixed prompts, and explicit model versioning.

A second limitation is that the configurator is designed specifically for BRKGA rather than for metaheuristics in general. That design choice is deliberate: random-key representations, decoder-centered search, and BRKGA’s elite-mutant-biased crossover structure provide a clean interface between problem semantics and executable algorithm design. However, this also means that our current mapping from natural-language problem descriptions to solver structure is expressed in a BRKGA-specific vocabulary. Generalizing the approach to other metaheuristics would require more than replacing the search engine. It would require support for different chromosome or solution encodings, different operator vocabularies, and different constraint-handling paradigms. For example, canonical genetic algorithms may require direct solution encodings and problem-specific crossover/mutation operators; local-search or neighborhood based metaheuristics would need move-set design instead of decoder synthesis; and other frameworks may rely on repair, penalties, feasibility rules, or adaptive memory rather than BRKGA-style decoder-based feasibility. Extending the configurator beyond BRKGA is therefore an important direction for future work rather than a claim of the current system.

We also emphasize that our choice of Claude Sonnet was a deliberate experimental design decision. We selected it because it is accessible through a stable API, supports explicit versioning for reproducibility, and has shown strong practical performance on code-generation tasks, making it appropriate for a study centered on

generating executable optimization components. At the same time, the results reported here should not be interpreted as model independent. Different LLMs may yield different configuration quality, different error modes, or different sensitivity to prompt structure. A systematic multi-LLM comparison is therefore an important direction for future work.

Finally, although the structured configurator reduces the risk of invalid architectural choices, it does not eliminate the need for careful validation. Our debugging episode showed that an LLM can produce code that appears plausible while still violating explicit requirements. The framework mitigates this risk by constraining generation through templates and predefined architectural patterns, but broader evaluation is still needed to determine how well these safeguards generalize across additional problem classes, models, and solver families.

6.4 When to Use Each Approach

The results suggest that the choice between LLM+Config and LLM-Only should be driven primarily by problem structure. For problems with simple constraint structure, such as standard TSP, we do not recommend LLM+Config. As shown in Table 2, both approaches solve the smallest instances optimally, and LLM+Config is only marginally better on some medium-sized cases. However, on the largest TSP instances the trend reverses: LLM-Only substantially outperforms LLM+Config, achieving gaps of 8.84% versus 23.87% on gr666, 9.65% versus 24.30% on rat783, and 8.61% versus 32.00% on dsj1000 (Table 2). This indicates that when a straightforward permutation decoder is sufficient, the framework's additional architectural machinery (including multi-component chromosomes, multi-pass simulation, and structured constraint handling) adds overhead without corresponding benefit.

In contrast, LLM+Config becomes preferable once the problem requires richer decoder structure or more tightly coupled decision components. This is already visible in the TSPJ results. In Table 3, LLM+Config achieves an average gap of 1.5% from optimal, compared with 2.9% for LLM-Only. More importantly, Table 4 shows that LLM+Config also slightly outperforms the specialized state-of-the-art MGA method, achieving a 1.5% gap from best known solutions versus 1.63% for MGA, despite requiring no manual operator design or extensive parameter tuning. These results suggest that even moderately more complex problems, such as TSPJ, benefit from the configurator's ability to select a more appropriate architecture automatically.

The case for LLM+Config is strongest for heavily constrained problems such as VRP-RPD, where feasibility, operation ordering, and decoder design are central. As summarized in Table 5, the original LLM-Only workflow is dramatically worse than LLM+Config, with an average makespan degradation of 149.75% among valid comparisons. Even after correcting the major two-phase scheduling flaw, LLM+Config still wins 43 of 57 comparisons and ties the remaining 14, losing in none. The corrected LLM-Only approach remains significantly worse in larger instances, with degradations such as 29.5% in kroA100, 30.9% in eil101 and 37.5% in gr202, while performance in the largest cases can further degrade after correction. These results show that when the problem includes strong precedence constraints, capacity constraints, or

multi-agent coordination, the framework's structured decoder construction provides a clear and consistent advantage.

Overall, the evidence from Tables 2–5 supports the following guideline: use LLM-Only for relatively simple problems where a lightweight, conventional decoder is sufficient, and use LLM+Config for problems whose semantics imply nontrivial architectural requirements, especially multi-component encodings, simulation-based decoding, or strong constraint interactions. Regardless of the workflow chosen, LLM generated code should still be validated carefully; however, the results indicate that structured configuration becomes increasingly valuable as problem complexity rises.

7 Conclusion

We have presented a framework combining LLM-based problem understanding with a modular BRKGA configurator. The key finding is that framework value scales with problem complexity: LLM-Only suffices for simple problems (TSP), while LLM+Config becomes essential as constraints increase (VRP-RPD, TSPJ).

On VRP-RPD, the original LLM-Only with two-phase scheduling averaged 150% worse than LLM+Config. After correcting the interleaving flaw, LLM-Only still lost or tied in all 57 comparisons. On TSPJ, LLM+Config achieved 1.5% gap versus 1.63% for the state-of-the-art MGA [2]. However, on unconstrained TSP, LLM-Only outperformed LLM+Config on large instances (8–9% vs 24–32% gap), demonstrating that framework overhead can hurt when its features provide no benefit.

Future work includes automatic problem complexity assessment to recommend approaches, and extending to continuous and multi-objective optimization.

Acknowledgments

Generative AI Disclosure: Claude (Anthropic) was used for: (1) drafting and editing portions of the manuscript, (2) developing components of the experimental code, and (3) the LLM-based problem analysis described in Section 3.1. All outputs were reviewed, verified, and modified by the authors, who take full responsibility for the content.

References

- [1] M. Mosayebi, M. Sodhi, and T. Wettergren, "The traveling salesperson problem with job-times (TSPJ)," *Computers & Operations Research*, vol. 129, p. 105226, 2021.
- [2] P. Gutiérrez-Aguirre and C. Contreras-Bolton, "A multioperator genetic algorithm for the traveling salesman problem with job-times," *Expert Systems with Applications*, vol. 240, p. 122472, 2024.
- [3] B. Romera-Paredes, M. Barekatin, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. R. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, et al., "Mathematical discoveries from program search with large language models," *Nature*, vol. 625, no. 7995, pp. 468–475, 2024.
- [4] S. Liu, C. Chen, X. Qu, K. Tang, and Y.-S. Ong, "Large language models as evolutionary optimizers," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–8, IEEE, 2024.
- [5] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Proceedings of the International Conference on Learning and Intelligent Optimization (LION)*, pp. 507–523, Springer, 2011.
- [6] M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, M. Birattari, and T. Stützle, "The irace package: Iterated racing for automatic algorithm configuration," *Operations Research Perspectives*, vol. 3, pp. 43–58, 2016.
- [7] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook, *The Traveling Salesman Problem: A Computational Study*, Princeton University Press, 2011.

- [8] G. Reinelt, "TSPLIB: A traveling salesman problem library," *ORSA Journal on Computing*, vol. 3, no. 4, pp. 376–384, 1991.
- [9] C. C. Sartori, M. I. Pino, P. Pinacho-Davidson, and C. Blum, "LLM-based instance-driven heuristic bias in the context of a biased random key genetic algorithm," arXiv preprint arXiv:2509.09707, 2025.
- [10] F. Da Ros, M. Soprano, L. Di Gaspero, and K. Roitero, "Large language models for combinatorial optimization: A systematic review," arXiv preprint arXiv:2507.03637, 2025.
- [11] A. Hottung, F. Berto, C. Hua, N. G. Zepeda, D. Wetzel, M. Römer, H. Ye, D. Zago, M. Poli, S. Massaroli, et al., "VRPAgent: LLM-driven discovery of heuristic operators for vehicle routing problems," arXiv preprint arXiv:2510.07073, 2025.
- [12] C. Yang, X. Wang, Y. Lu, H. Liu, Q. V. Le, D. Zhou, and X. Chen, "Large language models as optimizers," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.
- [13] M. Lindauer, K. Eggensperger, M. Feurer, A. Biedenkapp, D. Deng, C. Falkner, F. Hutter, and F. Hoos, "SMAC3: A versatile Bayesian optimization package for hyperparameter optimization," *Journal of Machine Learning Research*, vol. 23, no. 54, pp. 1–9, 2022.
- [14] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 24, pp. 2546–2554, 2011.
- [15] S. Ropke and D. Pisinger, "An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows," *Transportation Science*, vol. 40, no. 4, pp. 455–472, 2006.

A Supplementary Experimental Results

Complete per-variant results for all 60 instance-variant combinations, along with source code and additional experimental data, will be made available upon publication.

For LLM-Only without interleaving, LLM+Config wins all 60 comparisons with an average degradation of 149.75% for LLM-Only.

For the corrected LLM-Only with interleaving enabled, LLM+Config wins 43 of 57 comparisons (75%), ties 14 (25%), with an average degradation of 24.0% for LLM-Only.

B Prompt Templates

Both workflows received identical VRP-RPD problem descriptions that explicitly stated "INTERLEAVING: Operations can be interleaved, not 'all drops then all picks'." Despite this explicit requirement, LLM-Only generated a two-phase decoder. The LLM+Config workflow additionally receives BRKGA framework context, which enables correct unified priority ordering.

C Problem Definitions

For TSPJ, the input consists of n cities with distance matrix D and job time matrix $J[c][j]$. The decision variables are a city permutation π and job assignment σ . The objective is to minimize $\max(\sum_i D[\pi_i][\pi_{i+1}], \max_i (T_{\pi_i} + J[\pi_i][\sigma(\pi_i)]))$.

For VRP-RPD, the input consists of M agents, N customers, capacity K , distance matrix D , and processing times P . The constraints require that pickup occurs after dropoff plus processing time $P[c]$, that inventory remains in the range $[0, K]$, and that cross-agent pickup is allowed. The objective is to minimize makespan.