From Hand-Crafted to LLM-Based Variation Operators in Metaheuristics: A Tutorial

CAMILO CHACÓN SARTORI, Apeiron Intelligence, Spain and Artificial Intelligence Research Institute (IIIA-CSIC), Spain
GUILLEM RODRÍGUEZ-COROMINAS, Artificial Intelligence Research Institute (IIIA-CSIC), Spain and Universitat Politècnica de Catalunya (UPC), Spain
CHRISTIAN BLUM, Artificial Intelligence Research Institute (IIIA-CSIC), Spain

Large language models (LLMs) are increasingly being employed as variation operators in metaheuristics, generating or modifying candidate solutions, heuristics, or programs inside iterative search loops. This shift reframes variation as a model call conditioned on different types of information. We introduce an operator-level framework with two descriptors: (1) the type of prompt-conditioning information at variation time (`Numeric`, `Symbolic`, `Linguistic`), and (2) artifact persistence, identifying what survives the model call (`Transient`, `Amortized`, `Transfer`). The tutorial shows how to classify, build, and select these operators through a worked build template, a method survey, an evidence table, and a cost-aware decision guide.

Additional Key Words and Phrases: large language models, metaheuristics, variation operators, evolutionary computation, combinatorial optimization, prompt conditioning, artifact persistence

Paper website: https://camilochs.github.io/semantic-turn-metaheuristics

1 Introduction

Metaheuristic search is shaped by variation operators such as local search moves, mutation, recombination, or edit rules that generate new candidate solutions. In classical metaheuristics, such operators are usually hand-designed. Machine-learning integrations from past years often assisted the search process of metaheuristics by predicting objective function values (solution scores), selecting operators, or warm-starting solvers, while variation operators themselves remained fixed by the designer. A growing class of LLM-based methods changes this paradigm [59, 82, 97, 113]: an LLM generates or edits candidate solutions, heuristics, or programs directly at the variation step, inside an iterative search loop. We call this shift a *semantic turn*: variation becomes a model call conditioned on specific types of information at *variation time*, i.e., the time at which the variation operator is executed.

This semantic shift raises practical questions that existing taxonomies of variation operators do not address. Consider a practitioner developing an LLM-assisted Traveling Salesman Problem (TSP) metaheuristic search procedure: should the prompt include (a) only performance scores of candidate heuristics, (b) the current heuristic as executable code, or (c) a natural-language reflection on why the heuristic stalls? These choices can shift costs substantially across validation, evaluation, and LLM inference, so the right choice is domain- and budget-dependent. The literature contains many such methods, but lacks a shared operational framework for navigating among them.

This tutorial is written for researchers and practitioners in optimization—especially for those developing iterative, stochastic search processes such as metaheuristics—who want to beneficially incorporate LLMs into their methods. We assume familiarity with basic metaheuristic loops, but not with transformer architectures or LLM internals; Section 2 provides the LLM background

Authors' Contact Information: Camilo Chacón Sartori, Apeiron Intelligence, Barcelona, Spain and Artificial Intelligence Research Institute (IIIA-CSIC), Bellaterra, Spain, camilo.chacon@apeironagents.tech; Guillem Rodríguez-Corominas, Artificial Intelligence Research Institute (IIIA-CSIC), Bellaterra, Spain and Universitat Politècnica de Catalunya (UPC), Barcelona, Spain, guillem.rodriguez.corominas@upc.edu; Christian Blum, Artificial Intelligence Research Institute (IIIA-CSIC), Bellaterra, Spain, christian.blum@iiia.csic.es.

Table 1. Complementary positioning of this tutorial relative to recent surveys.

Work	Type of work	Organizing axis	After reading, you can…
LLM4AD [61] (CSUR)	survey	the LLM's *role*: optimizer / predictor / extractor / designer	locate a method by the role the LLM plays
Da Ros et al. [25] (CSUR)	systematic review (PRISMA)	task, architecture, dataset, application	find the census of LLM-for-CO (combinatorial optimization) studies
Wu et al. [105]	survey + roadmap	direction of enhancement: LLM-enhanced EA vs. EA-enhanced LLM	see how evolutionary computation (EC) and LLMs enhance each other
Zhang et al. [119]	survey	integration stage: modeling → solving	trace the modeling-to-solving pipeline
This tutorial	**tutorial**	what the operator is *conditioned on*: `Numeric` / `Symbolic` / `Linguistic`	**build, classify,** and **choose** an operator

needed here. We focus on variation operators conditioned through text prompts. Fine-tuned or adapter-based models remain in scope when the variation step is still prompt-conditioned. In those cases, we classify the prompt channel in the same way. By contrast, operators whose conditioning is encoded only in the model parameters or supplied through non-text inputs such as images fall outside our scope and are discussed as open directions in Section 8.

The organizing lens. We view existing applications through one structural question: *on what type of problem structure is the operator conditioned?* We distinguish three conditioning channels—`Numeric`, `Symbolic`, and `Linguistic`—and pair them with a second descriptor, *artifact persistence*, which describes how the emitted artifact is used within the search: consumed inside the current search loop, reused after offline design, or re-bound to a new instance family or domain. Together, the two descriptors form an operator-level taxonomy for practice: they allow us to classify methods, guide the construction of validated prompt-conditioned variation loops, and support family-level choices under evaluator, budget, and deployment constraints.[1]

The lens is a complementary teaching spine: Table 1 contrasts it with surveys organized by model role, the direction of interaction between optimization algorithms and LLMs, integration stage, or systematic-review metadata such as task, architecture, dataset, and application. Readers familiar with those surveys should find here a practical route through the operator-design problem: identify the load-bearing prompt content, implement a validated variation loop, and decide when the LLM belongs in the search pipeline.

Our contribution is threefold. First, we introduce a two-descriptor framework for LLM variation operators: dominant conditioning channel and artifact persistence. Second, we make the framework operational through a reusable build template with parsing, validation, and bounded repair. Third, we synthesize representative methods through that framework and give a cost-aware decision guide for choosing among them.

The paper is organized to mirror this workflow. Section 2 shows how an LLM call can be read as a stochastic variation operator. Section 3 defines the conditioning and persistence descriptors, grounds the term "semantic" in the classical syntax/semantics distinction, and gives the placement rule. Section 4 teaches the mechanics of building an operator, from the prompt template to the validator and the repair loop. Section 5 places representative methods on the lens, and Section 6

[1]Throughout, the lens is descriptive rather than predictive: it organizes the design space, but does not claim a universal law linking more explicit problem representations in the prompt to better optimization performance. Whether such structure helps remains an empirical, task-dependent question, which we return to in Section 8.

turns that map into a decision guide. Sections 7 and 8 collect the engineering risks and open research directions, including the reproducibility and reporting discipline needed for credible claims. The appendices provide a copyable prompt template, a Python reference loop, a channel-ablation audit, and an extended coverage map. To support practical adoption and reproducibility, we provide a repository[2] with the materials developed for this tutorial, including prompt templates, reference code, and other supplementary resources.

2 Background and preliminaries

This section introduces the background needed to view prompt-conditioned LLM operators as variation mechanisms: the structure of metaheuristic search loops, the mechanics of LLMs as conditional samplers, the proposal-distribution framing that connects the two, and related research fields that motivate the boundary of the classification framework that we introduce (later called *the lens*). In this context, note that—in accordance with LLM notations and expression—we often call the output of an LLM-based variation operator a proposal.

The metaheuristic search loop. A metaheuristic maintains a search state—an incumbent solution, a population of solutions, an archive, or a set of candidate artifacts—and repeatedly proposes, evaluates, and selects among alternatives [31, 94]. The artifact being varied may be a solution, a move rule, a heuristic, or a program, depending on the level at which search is being performed. We use a single-artifact schematic only to mark the proposal step that an LLM may replace:

```
x ← initial_solution()
best_seen ← x
while not stop_condition:
    x' ← Vary(x)                                          # the variation operator
    if f(x') better than f(best_seen): best_seen ← x'
    if Accept(x', x): x ← x'
return best_seen
```

Here Accept denotes the selection rule, which may be greedy or probabilistic; `best_seen` is an elitist record tracked separately because the current state can temporarily be worse than the best candidate found. The shift studied here replaces a hand-coded Vary — for example, bit-flip and 2-opt neighborhoods in stochastic local search [45], tabu-list restrictions [33], or Ant Colony Optimization construction rules [27] — with a prompt-conditioned LLM call. In that case, parsing and bounded repair occur logically between Vary and Accept: a candidate that cannot be repaired is rejected before the acceptance rule evaluates it. Because such calls are expensive in latency, token use, and API or deployment cost relative to classical moves, the call budget shapes when and how often the model should be invoked (Sections 4 and 6). Algorithm 1 later formalizes this schematic with the notation used in the build section.

An LLM is a conditional sampler. Several facts about LLMs are sufficient for this tutorial. A *prompt* is the text fed to the model. Text is represented as *tokens*, discrete units from the model's fixed vocabulary; at each generation step the model samples one token from a probability distribution over that vocabulary, extending the sequence until a stopping criterion is met (Fig. 1). A *completion* is the decoded sequence returned by the model.[3] *Temperature* controls randomness by flattening or sharpening this next-token distribution (higher is more diverse, lower more stable). Top-k and

[2] https://camilochs.github.io/semantic-turn-metaheuristics

[3] Many APIs also expose structured-output schemas, tool/function calling, or grammar-constrained decoding, which constrain the form of a completion before parsing; in this tutorial they are validation aids and generation-interface choices, not new conditioning channels [32, 102].

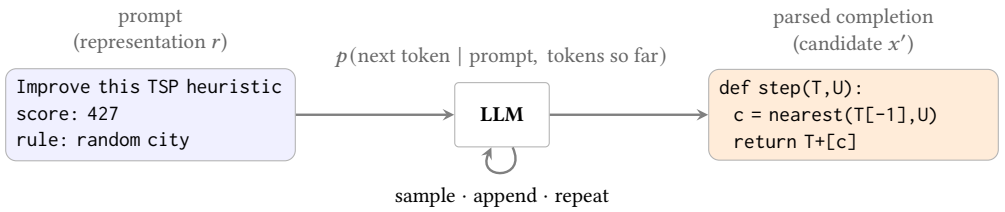

Fig. 1. Token-level view of an LLM call. A prompt containing the representation r is converted into a completion by repeatedly sampling a next token conditioned on the prompt and the tokens already generated. The completed text is then parsed as a candidate artifact x'.

top-p are common decoding filters over the next-token choices: top-k restricts sampling to the k most likely next tokens, while top-p restricts it to the smallest set of likely tokens whose cumulative probability mass reaches p. These decoding controls reshape the induced proposal distribution, analogous to how mutation rates or step sizes tune a classical stochastic operator.

In-context conditioning embeds task context, examples, or both in the prompt to induce task-specific behaviour without changing the model's weights[4] [11]. *Few-shot* conditioning is the special case where example input–output pairs are included; *zero-shot* conditioning provides only a task description or search-state representation, with no exemplar completions. One useful view of in-context learning is that the prompt acts as evidence about the task the model is being asked to perform, so examples and surrounding context can shift which completions are likely under the fixed weights [106].[5]

Prompt phrasing, ordering, and delimiters create format sensitivity: because the model predicts each next token conditioned on the exact preceding text, even substantively equivalent rephrasings can change which continuations are likely. Long prompts also add a placement issue: models often use information at the beginning or end of context more reliably than information buried in the middle [64]. For our purposes, the prompt is the designer's main interface to the sampler (Section 3): changing what it contains reshapes the distribution of all subsequent token choices.

The previous points describe a single completion; across a search run, feedback must be carried from one call to the next. At the API level, each LLM call is stateless: the model's weights do not change between calls.[6] Each call also has a finite *context window*, the maximum combined length of the prompt and completion that the model can process. Accumulated feedback must therefore be summarized or pruned when it exceeds that budget, introducing compression losses.

For search, the model's stochasticity — diversity across completions on repeated calls — is an asset: it drives exploration in the role of classical perturbation. This stochastic decoder becomes useful only after it is embedded in an operator. An LLM variation operator can be viewed as a **proposal distribution** $q(x' \mid r)$ that maps a representation r of the problem and current search state to a distribution over candidate artifacts x' (Fig. 2) — a notion already familiar from stochastic local search. In the schematic baseline, a classical operator applies a fixed or hand-designed move, neighborhood, mutation, or construction rule. In an LLM operator, the central design object is instead r: the explicit representation placed in the prompt at generation time. Changing r changes

[4]Those conditional probabilities come from pretraining: the model's weights encode regularities learned from large corpora of text and code, which shape the next-token distribution at inference time.

[5]Prompts can also ask the model to expose intermediate reasoning before producing the final artifact; the artifact tokens are then sampled conditional on that stated rationale, echoing chain-of-thought (*CoT*) prompting [101].

[6]Session-based APIs may accumulate conversational context server-side, but the reliable design assumption is that adaptation across search steps must be written into successive prompts.

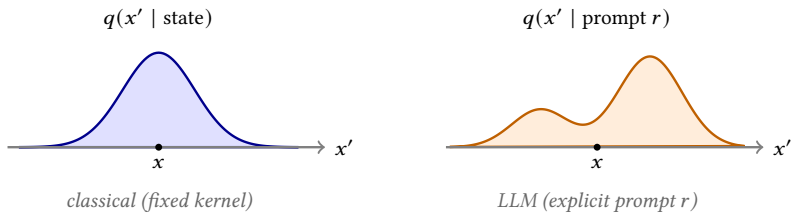

Fig. 2. The operator as a proposal distribution $q(x' \mid r)$. Left: a classical fixed kernel produces a narrow proposal centred on the current state x. Right: an LLM prompt can induce a different, sometimes broader distribution over candidate artifacts.

the induced proposal distribution and, therefore, the properties of the resulting artifacts —including whether they can be parsed, validated, executed externally, or compared.

Strictly, the model samples token sequences, not solutions or programs. Because next-token likelihood is not the same as domain feasibility or executable correctness, outputs can be malformed, infeasible, or non-executable. Parsing, validation, and *bounded repair* (a limited number of repair attempts after invalid output) map those sequences to candidate artifacts, rejecting unrepaired outputs before selection. Thus $q(x' \mid r)$ denotes the artifact-level proposal distribution induced by the whole variation module, not just the raw next-token distribution of the LLM. Unless stated otherwise, the notation suppresses fixed choices such as model version, decoding settings, parser, validator, and repair policy. Both classical and LLM operators are treated here as stochastic proposal mechanisms; nothing in this framing requires the model to "understand" anything.

The notation is useful because it names measurable design effects. Changing r can alter the probability that raw completions parse and validate, the number of repair attempts per accepted artifact, the locality or diversity of proposals, and the probability mass near feasible high-quality artifacts. High invalid-output or repair rates are therefore diagnostic: the model's raw completions have poor overlap with the feasible artifact space, so r is not steering the variation module toward valid candidates.

From an optimization perspective, this recalls Estimation of Distribution Algorithms (EDAs) [55], which explicitly learn and sample candidate distributions; here the artifact-level distribution is induced implicitly by prompt content, decoding, parsing, validation, and repair. This analogy is only a bridge for optimization readers: LLM operators do not inherit EDA modeling assumptions or formal properties.

Boundaries with related literature. Prompt-conditioned variation is a continuation of a long-running integration of machine learning into metaheuristics, with a deep-learning wave accelerating over the past decade, including *ML into metaheuristics* [23, 50, 95], *learnheuristics* [15], *ML for combinatorial optimization* [7], and hybrid metaheuristics [8]. Some LLM-hybrid work remains assistive: it extracts instance patterns that inform a conventional metaheuristic rather than generating the variation itself [18]. Here, we focus on the point where the prompt-conditioned model call becomes part of the variation operator itself. The boundaries below clarify how that prompt interface differs from concepts in related fields.

Automated algorithm design is the closest relative to LLM-based automatic heuristic design. In this part of the literature, the model proposes or revises the heuristic, operator, or program that will later be run by the solver. Two boundary cases delimit the space. Algorithm configuration [47, 68] searches over parameters of a fixed algorithm structure; an LLM operator reduces to configuration

when the prompt carries parameter proposals but does not change the operator family (the underlying move class, heuristic template, or solver component being tuned). Algorithm selection [53, 81] chooses among a predefined portfolio using problem or runtime features; an LLM-based routing method is a prompt-conditioned instantiation of selection when it selects among fixed operators. Code-emitting operators that synthesize new heuristics or programs sit at the design end of the spectrum.

Hyper-heuristics [13, 14] also operate on a higher level, selecting or generating heuristics and, in cross-domain forms, transferring control strategies across problem families. LLM-based variants inherit that ambition, but add prompt-readable histories, code, and rationales as conditioning signals; unlike many selector-style hyper-heuristics, they need not choose only from a predefined library of low-level heuristics.

In Genetic Programming (GP), Geometric Semantic Genetic Programming (GSGP) [76, 98] already gives "semantic" a behavioural meaning: variation can be defined with respect to program output rather than only tree syntax. Here the object being varied may still be a program, but the distinctive move is different: GSGP uses algebraic variation with formal structure in output space, whereas an LLM operator samples from an implicit prompt-conditioned distribution without algebraic closure or a fixed semantic metric.[7]

Bayesian optimization [88] is another nearby tradition for expensive black-box objectives: it learns an explicit surrogate model and acquisition rule over the objective landscape, whereas an LLM variation operator changes the proposal mechanism induced by the prompt. The distinction matters because BO uses learning to choose where to evaluate the objective next, whereas the LLM call is itself the variation operator that generates the next candidate artifact.

Neural combinatorial optimization [52] is also related but structurally distinct, training learned models to construct or guide solutions directly. Our lens applies when the search-relevant conditioning signal is exposed in the text prompt; when that signal is absorbed into model weights, the method falls outside the prompt-conditioning view. In each case, the boundary is the prompt interface: in related fields either the operator structure is fixed, learning is done without readable prompt conditioning, or semantics are defined over representations rather than over text supplied to the LLM at variation time.

These boundaries leave the central design question intact: once the proposal step is a prompt-conditioned LLM call, classification depends on which prompt-readable signal conditions the next candidate and which artifact, if any, survives the call. The next section turns these two attributes into the conditioning–persistence lens.

3 The organizing lens: conditioning and persistence

Section 2 has identified the prompt representation r as the design variable in an LLM proposal distribution. This section turns this variable into the paper's organizing lens. We first provide rules for determining the dominant conditioning channel in the variation prompt of an existing method, namely, the prompt signal that most strongly shapes the next candidate artifact. Moreover, we add a persistence descriptor for what survives after the LLM call. Together, conditioning and persistence form the descriptive framework used throughout the rest of the paper.

A dominant-conditioning typology. We classify methods by the dominant load-bearing channel in the prompt at the variation step (Fig. 3). `Numeric` conditioning consists of the encoded search information familiar from classical metaheuristics: solution encodings, instance features, scalar scores, ranked parents, or score trajectories. `Symbolic` conditioning consists of machine-interpretable artifacts with formal structure—code, abstract syntax trees, formal rules, or structured

[7] Section 3 separates this behavioural use of "semantic" from the prompt-channel use developed here.

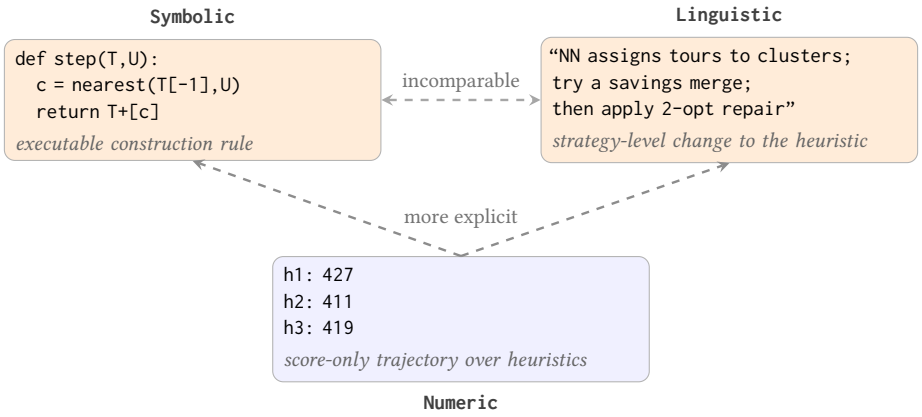

Fig. 3. The conditioning lens made concrete: one TSP operator (“propose the next construction heuristic”) conditioned on numeric, symbolic code, or linguistic content. These alternatives change the prompt representation r supplied at the variation step and therefore shape the induced proposal distribution shown in Fig. 2.

graphs—whose syntax or behaviour can be checked by an external parser, validator, compiler, or executor. `Linguistic` conditioning consists of operative natural language (NL): dynamic critiques, reflections, diagnoses, design principles (which qualify even when reused across iterations, because their content encodes search-derived knowledge), or strategy notes that steer later proposals.[8]

Hybrid methods and dominant channels. We regard the category `Numeric` intentionally in a broad way: a scalar score, a trajectory, and a feature vector all fall into `Numeric`, although they expose different amounts and kinds of search information. They share an evaluative role: they answer how candidate artifacts perform, although different summaries can affect behaviour differently without changing the conditioning class. `Numeric` is the base channel because it supplies the encoded state and objective feedback that metaheuristics already consume; `Symbolic` and `Linguistic` differ not only in surface form but in checkability: `Symbolic` content exposes behaviour that can be parsed, executed, or validated, whereas `Linguistic` content carries propositional or strategic rationale that can guide search but is not mechanically verified in the same way. Both are more explicit about problem structure than `Numeric`, but they are *not* ranked against each other: code can be more structure-preserving than prose, while prose can express abstractions not present in the code. Most real methods are **hybrids** that use several channels at once. We therefore report the full set of channels present and classify a method by its *dominant* one — the channel that most drives the proposal.

The analogy to black-box, gray-box, and white-box optimization is informative but not identical [86]: Santana’s axis ranks how much of the *problem’s* structure is available, whereas our axis classifies the type of information placed in the LLM prompt. *FunSearch* illustrates the distinction. The LLM receives prior program attempts selected by score, so the process is closer to gray-box than a pure scalar-only loop, yet on our axis it is `Symbolic` because the dominant conditioning signal is program code.

A reproducible rule for operator classification. A method’s channel assignment is an operational label for design intent: it identifies which prompt signal is intended to drive variation. The rule consists of two steps. First, identify the prompt’s operative, search-state-derived content. The

[8]Fixed task headers and static problem statements do not by themselves make a method `Linguistic`.

operative-not-boilerplate test filters out fixed headers and static problem statements: a header such as `[Task: Solve TSP]` does not lift a method to `Linguistic`, but a current reflection, diagnosis, or reusable principle can. Second, identify the channel intended to carry the operative search signal and audit that classification, where possible, by a *drop-channel* ablation: remove or neutralize one channel at a time and ask which removal most changes the induced proposal while preserving a valid scaffold. The classification is therefore determined by the prompt signal that steers variation, not merely by the kind of artifact the method emits. When `Numeric` appears alongside operative `Symbolic` or `Linguistic` content, it is recorded as a supporting base channel. It becomes dominant only if that more explicit content fails the operative-not-boilerplate test. Likewise, a code artifact can be required for parsing or execution without being the dominant signal if the method's distinctive variation step is steered by an operative natural-language idea or reflection. If an ablation is inconclusive, report the source-described design-intent channel with the caveat; if two non-base channels both materially change proposals and neither clearly dominates, report a hybrid rather than forcing a single channel classification.

This ablation is an audit of the classification, not a clean causal attribution test. LLM outputs are sensitive to exact prompt phrasing, ordering, delimiters, decoding settings, and interactions among channels (Section 2). Single-channel ablations can also miss complementarities between channels, so the rule is a placement audit and information-signal diagnostic, not an estimate of independent causal contribution. The procedure therefore classifies the researcher's intended conditioning signal rather than inferring which tokens the model internally weights most. Assignments are reproducible for a given task and operator configuration; a different implementation could receive a different classification. Section 5 applies this rule in the method survey. Appendix B adds a description-level agreement check: three LLM-based coders — one instance each from three different model families — classified the twelve representative methods of Table 2 from their channel descriptions, agreeing unanimously on nine of the twelve methods (Gwet's AC1 [39] = 0.73). The disagreements occur where channel dominance is least determinate, including a case already marked as a boundary case. This is a reliability check for the tutorial's representative classifications, not a claim that every method in the broader literature is equally unambiguous.

As a worked example, consider a ReEvo-style generator–reflector loop [113]. The variation prompt contains the current code, its score, and a natural-language reflection about why the code should change. The code is necessary: dropping it removes the executable artifact to be edited. But the distinctive operative signal is the reflection: neutralizing it reduces the method toward code-only automatic heuristic design, while preserving the scaffold and score feedback. The classification is therefore `Linguistic`-dominant with `Symbolic` support. This example demonstrates the classification procedure; it does not claim that the reflection channel is causally necessary for performance on every task.

Where the term "semantic" comes from. The word "semantic" follows the classical syntax/semantics contrast: syntax is surface form, while semantics is the meaning a representation carries [90, 103]. Classical variation often manipulates the syntactic form of an encoded state through a hand-designed move, neighborhood, mutation, or construction rule; its domain knowledge is built into the operator's logic. Prompt-conditioned LLM variation changes that interface by exposing problem-relevant meaning in r, the prompt representation that conditions the next candidate. The three channels then name different kinds of such meaning. Numeric conditioning carries *evaluative* meaning — "how good is this?" `Symbolic` conditioning carries *behavioural/denotational* meaning — "what does this compute or do?"[9] `Linguistic` conditioning carries *propositional* meaning —

[9]This is the sense closest to Geometric Semantic Genetic Programming (GSGP), where semantics is program behaviour rather than syntactic form [76, 98].

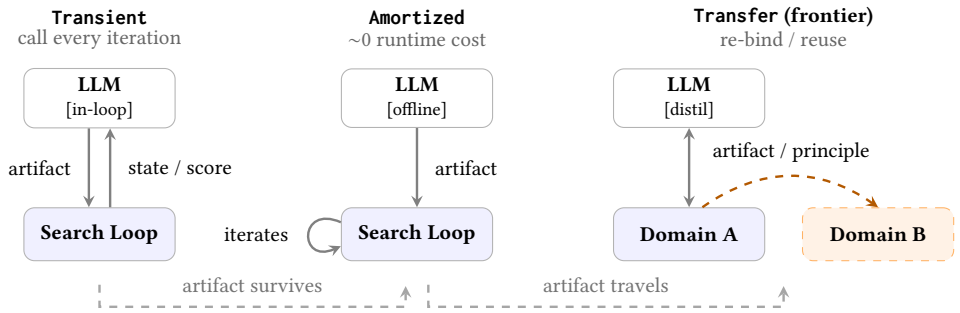

Fig. 4. What persists, as a descriptive attribute. `Transient`: the LLM remains inside the loop. `Amortized`: an offline call emits an artifact that later runs at near-zero runtime cost. `Transfer`: a source-domain artifact or principle is reused in a new domain without rediscovering it.

"why does this work, and what should change?" Therefore, the tutorial extends the syntax/semantics contrast from behavioural `Symbolic` content to evaluative `Numeric` and propositional `Linguistic` prompt content. It is not a claim that the model understands the problem, nor an NLP-style embedding-similarity claim.

A second descriptor: persistence. Persistence — what comes out and survives the call — is conceptually independent of conditioning, though the two are empirically correlated in current methods (Fig. 4). A `Transient` artifact is used within a single loop iteration; the loop therefore pays inference latency and token/API cost whenever it asks the model for a new candidate. An `Amortized` artifact is emitted offline and then runs at near-zero runtime inference cost, as a reusable operator, heuristic, or program. The frontier case is `Transfer`: an inspectable artifact or principle distilled from source-domain evidence is reused in a target domain without repeating the discovery process. `Transfer` is not new in spirit—classical cross-domain hyper-heuristics moved selection strategies across domains without an LLM [13]; the LLM-specific question is whether generated artifacts or principles are inspectable and re-bindable enough to travel robustly.

Persistence therefore bundles two practical subproperties: the *call-locus* (is the LLM in the loop or called offline?) and the *portability* of the surviving artifact. They usually align, but not always: executable artifacts may be re-bound directly, whereas prompt-mediated principles may still require model calls when applied to a new domain. In executable transfer, the artifact itself can run after its interface is adapted to the target evaluator. In principle transfer, the surviving object is a verbal rule or design rationale that seeds a new prompt and must then be validated on the target task.

Conditioning and persistence together characterize the methods surveyed in Sections 5 and 6. They are separate descriptors but often coupled in current methods: solution-level outputs tend to be `Transient`, operator or program outputs tend to be `Amortized`, and principle-like outputs are natural candidates for `Transfer`. These are tendencies, not constraints. Section 4 now instantiates the descriptors in a runnable prompt-conditioned operator.

4 Developing LLM variation operators

This section turns the descriptors of Section 3 into a runnable prompt-conditioned operator. The construction has four components: prompt assembly, sampling, parsing and validation with bounded repair, and loop integration. By the end of the section, the reader has a runnable scaffold; the full copyable prompt template and a Python reference loop are in Appendix A.

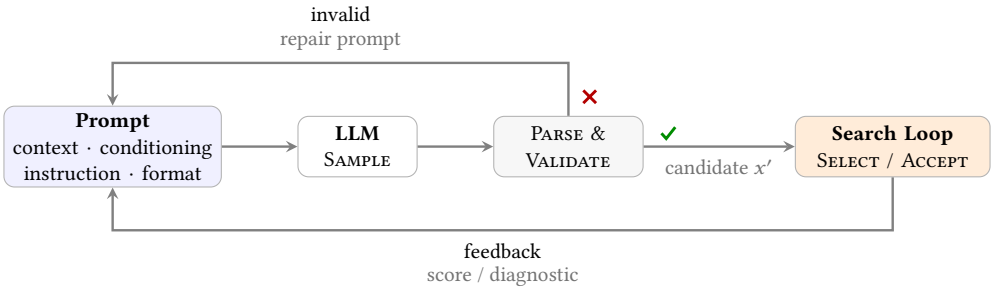

Fig. 5. Anatomy of an LLM variation operator: prompt, sample, parse-and-validate (with a repair path for invalid output), and loop integration with an optional feedback channel.

Anatomy. An LLM variation operator has four parts (Fig. 5): a *prompt* (problem context · conditioning content · instruction · output format); the *sample* call; a *parse-and-validate* stage that checks schema correctness and feasibility, returning a diagnostic on failure; and the *loop integration* that feeds accepted candidates to selection and, optionally, passes feedback (a score or a diagnostic note) into the next prompt. Algorithm 1 formalizes this loop.

Algorithm 1 formalizes the reference loop. The inputs (see **Require**; line 1) are the initial artifact x_0, the objective function f, the representation R and feasibility constraints C that together define a valid candidate, a step budget T, a repair cap K, and a stopping rule Stop. Here T is the maximum number of main-loop steps, not the number of accepted candidates or repair attempts: each step assembles one prompt, may spend up to K+1 LLM samples because of bounded repair, and counts as one step against T whether or not a candidate is accepted. The lowercase r is the assembled prompt representation used in $q(x' \mid r)$; the uppercase R is the candidate representation or schema used by the validator.

The first statements initialize the incumbent x_{cur}, the best-so far x_{best}, their cached scores, the history list H, and the step counter t. The history H stores validator diagnostics and selection outcomes; in the template below, it fills the "Scores and diagnostics" slot. The outer loop runs until the step budget is spent or Stop is triggered. The stopping rule represents the usual metaheuristic termination condition, such as a time limit, target quality, convergence test, or user interruption; any dependence on the initial artifact, wall-clock time, or a strict call cap can be captured inside Stop. Here t counts main-loop steps, while k indexes repair attempts within the current step.

Each step first assembles the prompt representation r from the current state: the representation, the constraints, the incumbent and its cached score, and the history gathered so far. The bounded-repair inner loop then samples a raw completion out from the LLM, and Validate parses and checks the output, returning a valid artifact y only on success. On success the operator leaves the inner loop. On failure, RepairPrompt resends the current prompt together with the failing output and the validator's diagnostic before re-sampling, for at most K+1 attempts. Because each failed attempt can add text to the repair prompt, implementations should cap or summarize the repair block when context grows (see Section 7). If the inner loop exhausts its retries without a valid candidate, the step is logged as invalid and the validator diagnostic is appended to H. The step is then skipped: it consumes one budget step but does not accept a candidate.

For a valid candidate y, the objective computes s_y, and Δ_y records its score gap to the incumbent. The selection rule may depend on the step counter t for any schedule or rule that changes over time; greedy acceptance can ignore that argument. An accept replaces the incumbent and its cached score, records the accepted candidate's score and gap, and updates the best-so-far from the

Algorithm 1 LLM variation operator with bounded repair

Require: x_0, f, R, C, step budget T, repair cap $K \geq 0$, stopping rule STOP

$x_{\text{cur}} \leftarrow x_0$; $s_{\text{cur}} \leftarrow f(x_{\text{cur}})$
$x_{\text{best}} \leftarrow x_{\text{cur}}$; $s_{\text{best}} \leftarrow s_{\text{cur}}$
$H \leftarrow [\,]$; $t \leftarrow 0$
while $t < T$ and not STOP$(x_{\text{cur}}, x_{\text{best}}, H, t)$ **do**
  $r \leftarrow$ ASSEMBLEPROMPT$(R, C, x_{\text{cur}}, s_{\text{cur}}, H)$
  for $k = 0, \ldots, K$ **do**
    out $\leftarrow$ SAMPLELLM(r)
    $(ok, y, err) \leftarrow$ VALIDATE(out, R, C)
    if ok **then**
      break
    end if
    $r \leftarrow$ REPAIRPROMPT(r, out, err)
  end for
  if not ok **then**
    $H \leftarrow H \,\|\, [(\text{invalid}, err)]$
    $t \leftarrow t + 1$
    continue
  end if
  $s_y \leftarrow f(y)$
  $\Delta_y \leftarrow s_y - s_{\text{cur}}$
  if ACCEPT$(y, s_y, x_{\text{cur}}, s_{\text{cur}}, t)$ **then**
    $x_{\text{cur}} \leftarrow y$; $s_{\text{cur}} \leftarrow s_y$
    $H \leftarrow H \,\|\, [(\text{accepted}, s_y, \Delta_y)]$
    if s_{cur} better than s_{best} **then**
      $x_{\text{best}} \leftarrow x_{\text{cur}}$; $s_{\text{best}} \leftarrow s_{\text{cur}}$
    end if
  else
    $H \leftarrow H \,\|\, [(\text{rejected}, s_y, \Delta_y)]$
  end if
  $t \leftarrow t + 1$
end while
return x_{best}

accepted incumbent when s_{cur} is better than s_{best}. A reject records the candidate score and gap, so the next prompt can use both successful and failed attempts as feedback. This feedback provides a mechanism for adaptation across steps rather than treating each proposal as independent of prior search outcomes. In either case, the outer budget counter is increased. On termination, the loop returns the best artifact found.

Output schema and validator. The VALIDATE call inside the bounded-repair loop is itself a structured procedure; Algorithm 2 specifies it before we instantiate the prompt template. The validator turns the output contract into executable checks and returns a compact diagnostic when a check fails. Here name(R) is the expected value of the `representation` field for the chosen representation R, e.g. `permutation` or `python`.

Algorithm 2 VALIDATE(out, R, C)

$m \leftarrow$ PARSEENVELOPE(out)
if m is incomplete **then**
  return $(false, \bot,$ "schema error")
end if
if m.representation does not match name(R) **then**
  return $(false, \bot,$ "representation mismatch")
end if
$y \leftarrow$ PARSEPAYLOAD(m.payload, R)
if y is parse error **then**
  return $(false, \bot,$ "syntax error")
end if
if not FEASIBLE(y, C) within timeout **then**
  return $(false, \bot,$ EXPLAINVIOLATIONS$(y, C))$
end if
return $(true, y, \emptyset)$

Algorithm 2 makes the three layers explicit as a sequence of guarded checks. The procedure stops at the first failed layer and returns a diagnostic specific to that layer. The envelope and representation checks run first: PARSEENVELOPE extracts the structured candidate record from the raw completion using the envelope format declared by the prompt template; an incomplete record produces a *schema error*, and an answer labeled `python`, for example, is rejected before a permutation parser tries to read it. Payload parsing follows: PARSEPAYLOAD converts the payload — the raw artifact text inside the envelope — into a typed artifact y according to R, e.g. a permutation, an assignment, or a Python function, while malformed payloads return a *syntax error*. Feasibility is checked last: an infeasible candidate returns concrete violations, e.g. a duplicated city or a capacity violation, which is far more useful to a repair prompt than a generic "invalid." Only a candidate that clears all three layers is returned as valid. A timeout is treated as a feasibility failure; its limit should be set relative to the evaluator budget so that one malformed candidate cannot consume the run. The ordering is deliberate: cheap structural checks run before expensive feasibility checks, and because each layer emits its own diagnostic, the REPAIRPROMPT call in Algorithm 1 can target exactly the failure that occurred. For code artifacts, PARSEPAYLOAD can perform syntactic checks such as abstract-syntax-tree (AST) parsing and import allow-listing. These checks reject malformed code and obvious undeclared dependencies, but they are not a security sandbox and do not rule out unsafe behavior through allowed names, dynamic execution patterns such as `eval` or `exec`, unsafe deserialization, or allowed modules that expose file, network, or system access. Execution-time safety belongs in FEASIBLE: code should be type-checked and run in a separate process or container with restricted builtins, blocked file/network access, and time and memory limits. For direct solutions, the same FEASIBLE layer applies the domain-specific constraint checker defined by C.

A reusable prompt template. With the loop and validator specified, the template defines what the designer fills in for each instantiation. It instantiates the prompt block of Fig. 5, while Algorithms 1 and 2 specify the sampling, validation, repair, and loop-integration blocks. The prompt has four sections and can be instantiated for solution proposals, code mutations, and heuristic design; the section roles stay fixed, while the representation R, constraints C, context, and feedback are filled with domain-specific content.

```
[context]
Problem type:
Objective direction: minimize | maximize
Candidate artifact: solution | operator | program
```

```
Available primitives: <functions, encodings, libraries, or moves>

[conditioning]
Representation R: <syntax of a candidate>
Constraints C: <hard feasibility rules>
Incumbent or parents: <artifact(s)>
Scores and diagnostics: <local evaluator output, failures, diversity notes>
Structural notes: <problem decomposition, known bottlenecks>

[instruction]
Emit exactly one new candidate artifact. Keep the representation R.
Use only the listed primitives where possible.
If diagnostics are present, target them.

[format]
Return only:
CANDIDATE
id: <short_id>
artifact_type: solution | operator | program
representation: <declared_representation>
payload:
<candidate in representation R>
END_CANDIDATE
```

Here the `CANDIDATE`/`END_CANDIDATE` markers are the concrete envelope convention that ParseEnvelope expects; an implementation using JSON or structured output would replace only this format convention and its parser. The `payload` field contains the candidate itself. The following two traces instantiate this template in different ways: the first instantiates R as a direct TSP permutation, so the payload is a solution (a tour); the second instantiates R as Python code for a bin-packing heuristic, so the payload is a program.

One traced iteration. The following is a simple illustrative build trace. It instantiates the framework as a `Numeric`-conditioned, `Transient` operator: the prompt contains a tour and score, the LLM proposes a successor solution, and the artifact is consumed inside the loop. Consider a five-node Euclidean TSP with coordinates $0 = (0,0)$, $1 = (1,0)$, $2 = (2,0)$, $3 = (2,2)$, $4 = (0,2)$, incumbent tour `[0,2,3,4,1]`, and evaluated tour length 9.236. Fig. 6 summarizes the exchange. The listings show the prompt, sampled output, validation, repair, and acceptance steps for one outer iteration of Algorithm 1.

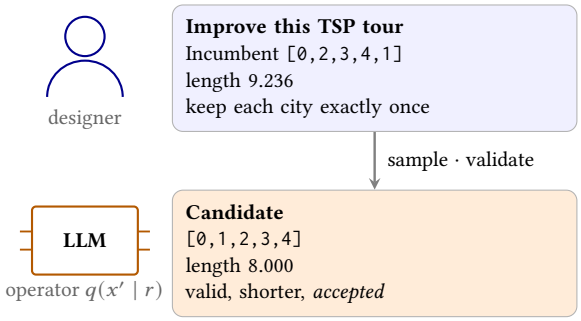

Fig. 6. An illustrative human–operator exchange: the *designer* authors the prompt, and the *LLM operator* samples a candidate that passes validation and improves the objective value.

```
PROMPT SENT
[context] Illustrative TSP instance; minimize Euclidean tour length.
[conditioning] R = permutation of [0,1,2,3,4] starting at 0.
Node coordinates: 0:(0,0) 1:(1,0) 2:(2,0) 3:(2,2) 4:(0,2).
```

```
C = every city exactly once; closed tour length is evaluator score.
Incumbent: [0,2,3,4,1], score 9.236.
[instruction] Emit one lower-length tour if possible.
[format] Use the CANDIDATE envelope.

SAMPLED OUTPUT
CANDIDATE
id: t1
artifact_type: solution
representation: permutation
payload:
[0,1,4,4,2]
END_CANDIDATE

VALIDATE
schema: ok; representation: permutation (match); payload: ok
feasibility: fail; city 4 duplicated and city 3 missing
```

Reading the first listing top to bottom: AssemblePrompt in Algorithm 1 produces the Prompt Sent block — the four sections of the template filled in for this instance. SampleLLM returns the Sampled Output: a well-formed `CANDIDATE` envelope whose payload is `[0,1,4,4,2]`. Validate then runs Algorithm 2: the envelope and declared representation pass, and the list parses, but the feasibility layer finds city 4 duplicated and city 3 missing, so it returns the diagnostic "city 4 duplicated and city 3 missing." This is a *schema-valid but infeasible* candidate: *ok* is false, the inner loop does not break, and RepairPrompt resends the full original context and appends the failing payload plus diagnostic:

```
REPAIR PROMPT (original context prepended; repair block shown)
The previous payload:
[0,1,4,4,2]
failed validation: city 4 duplicated and city 3 missing.
Emit one corrected CANDIDATE envelope. Do not change the representation.

REPAIRED OUTPUT
CANDIDATE
id: t1_fix
artifact_type: solution
representation: permutation
payload:
[0,1,2,3,4]
END_CANDIDATE

VALIDATE
schema: ok; representation: permutation (match); payload: ok
feasibility: ok
```

The second attempt (k=1) re-enters the inner loop: SampleLLM now returns the repaired output with payload `[0,1,2,3,4]` in the `REPAIRED OUTPUT` block. The `VALIDATE` block shows that the envelope, representation, payload, and feasibility checks now pass, so Algorithm 2 returns $(true, y, \emptyset)$ and the inner loop breaks. The objective scores the candidate at length 8.000 on this illustrative instance; since $8.000 < 9.236$, the acceptance rule returns true: the incumbent x_{cur} becomes the repaired tour, the best-so-far x_{best} is updated from that incumbent, and `(accepted, 8.000, -1.236)` is appended to the history H, so the next prompt is conditioned on the improvement. Had the repaired score been no better than 9.236, the candidate would still be valid but rejected by selection, and the history would record the same gap under a rejected label. One outer step of Algorithm 1 thus consumes, in this trace, two LLM samples and one accepted candidate.

A second trace: a code-emitting operator. The same loop can also instantiate a structurally different operator: a `Symbolic`, `Amortized` one that emits a reusable heuristic rather than a solution (Section 3). For online bin packing (bin capacity 10), the operator emits a Python `priority(item,`

`bins`) that ranks the open bins, and the evaluator scores a heuristic by the total number of bins it uses over a fixed set of instances with a lower bound of 19 bins. Starting from a first-fit heuristic (22 bins, 15.8% above the bound), one outer iteration conditions the prompt on that parent program and its score:

```
PROMPT SENT
[context]
Problem type: online bin packing
Bin capacity: 10
Candidate artifact: program
Available primitives: arithmetic, comparisons, enumerate, list comprehension, float("-inf")

[conditioning]
Representation R: python function priority(item, bins)
Constraints C: return one numeric priority per open bin; no imports
Parent program:
def priority(item, bins):
    return [1.0/(i+1) if b >= item else float("-inf")
            for i, b in enumerate(bins)]
Score from external evaluator: 22 bins; lower bound 19
Diagnostic from external evaluator: first-fit leaves avoidable residual capacity

[instruction]
Emit one replacement priority function that prefers feasible bins likely
to reduce the total number of bins. Preserve the signature exactly.

[format]
Return only:
CANDIDATE
id: <id>
artifact_type: program
representation: python
payload:
<python function>
END_CANDIDATE
```

The LLM then samples a candidate program and validates it — here ParsePayload parses the payload into an AST and rejects imports, and Feasible runs it in a restricted namespace (Algorithm 2) — before the evaluator scores it:

```
SAMPLED OUTPUT (LLM)
CANDIDATE
id: h1
artifact_type: program
representation: python
payload:
def priority(item, bins):
    # best-fit: prefer the tightest valid bin
    # infeasible bins receive -inf priority
    return [-(b - item) if b >= item else float("-inf") for b in bins]
END_CANDIDATE

VALIDATE (external)
schema: ok; representation: python (match); payload (AST): no imports
feasibility: ok (returns one score per input bin)

EVALUATE (external)
total bins = 19  (gap 0.0%, lower bound reached)  ->  accepted
```

The emitted best-fit heuristic reaches the lower bound for this illustrative instance set (19 bins) and is kept; it then runs with *no* further LLM calls — the cost is `Amortized`, in contrast to the per-move cost of the TSP operator above. Both traces are illustrative examples constructed in advance; their execution can be reproduced with the companion code released with this tutorial.

Design choices. The template leaves several choices to the designer; these should be reported because they determine the conditioning channel and persistence descriptor of Section 3.

- **In-loop vs. offline.** Calling the LLM every iteration gives a `Transient` operator: it is often simpler to implement and can adapt to the current search state, but incurs one model call per move. Emitting an artifact offline gives an `Amortized` operator: it front-loads the model cost, and is preferable when a reusable artifact can be evaluated cheaply many times and then deployed without runtime model calls.
- **Model choice.** The model is part of the operator design. Open-weight models are easier to pin, self-host, and run on private instances; API or private models may provide stronger completions but add version drift, data-exposure concerns, and per-token cost. Smaller models are generally cheaper and faster, whereas larger models often improve one-shot language-model quality and may reduce malformed completions in general settings [1, 43, 49]; for an operator, the trade-off should be measured on the target schema. The choice also couples to persistence: transient operators pay per call, while offline-amortized operators can spend a stronger model during design and deploy the emitted artifact without model access.
- **Feedback.** No feedback gives near-independent samples; score feedback gives `Numeric` conditioning; diagnostics or reflections can add `Linguistic` conditioning. Use richer feedback when diagnostics are reliable, but cap or filter it when noisy messages could steer the model toward spurious fixes or exhaust the context window.
- **Prompt layout.** Keep operative conditioning content close to the `[instruction]` slot rather than buried in a long `[context]` block; if the history grows, summarize or repeat the load-bearing signal near the end of the prompt (Section 7).
- **Output level.** Direct solutions are easy to parse and validate but usually transient. Programs and operators are more reusable and often `Symbolic`, but require a parser or compiler, sandboxed execution, unit tests, and resource limits.
- **Selection.** Classical acceptance keeps the evaluator auditable and cheap. An LLM-based evaluator adds model calls and can introduce model bias; use it only when objective evaluation is unavailable or intentionally part of the task.

The same scaffold also exposes recurring failure modes: invalid outputs, diversity collapse, token and repair growth, context overflow, prompt sensitivity, and benchmark exposure. These are not separate operator families but cross-cutting risks of prompt-conditioned variation. Section 7 collects the corresponding mitigation and reporting discipline, while Section 8 treats contamination and benchmark design as open infrastructure problems.

5 Classification of representative methods according to the lens

After demonstrating how different LLM-assisted operators can be built, we now classify representative published methods through the lens outlined in Section 3. Following the spirit of reference surveys in the field of metaheuristics [9], we classify methods, give representative ones a common skeleton, and place them in a shared frame. We first consider `Numeric`, `Symbolic`, and `Linguistic` operators, then discuss boundary and hybrid cases and map representative methods in Fig. 7.

How we select and incorporate methods. Our selection is *pedagogical, not systematic*. We do not aim to provide a complete census of the literature; recent systematic reviews, including a PRISMA review of this area, already serve that purpose [25]. Instead, we include a method in case at least one of the following aspects is fulfilled: (i) it can be classified and mapped to a *distinct point on the conditioning lens*, so the channels are populated by real instances; (ii) it introduces a *distinct*

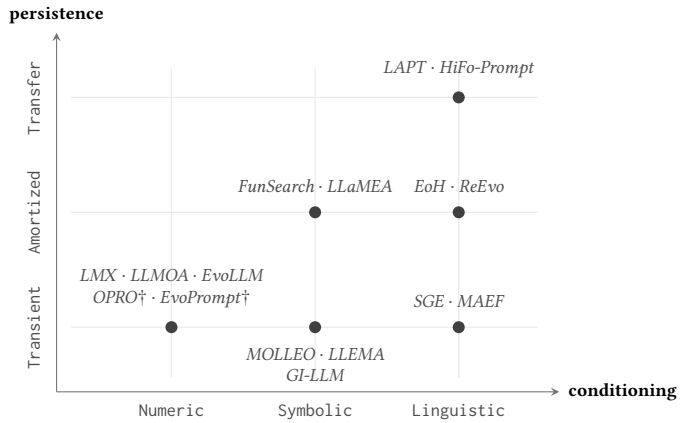

Fig. 7. Representative method classifications on the conditioning–persistence map (the lens) of Section 3. The horizontal axis records what conditions the LLM call; the vertical axis records what survives it. The † symbol marks prompt-optimization boundary cases whose placement depends on prompt design.

mechanism worth learning about (island population, reflection, (1+1) mutation, tree search, multi-objective selection); or (iii) it serves as a *background anchor* a newcomer needs, such as ELM, an open-ended-evolution precursor treated here as context rather than as a recommendation for scalar-objective optimization [56].

Every method we discuss is read against its *primary source*, so the skeletons (Algorithm 3) and the classifications (Fig. 7) are faithful to those sources; in those cases in which various methods from the literature can be mapped to the same place on the organizing lens, we deal with representatives only and point to the systematic review for the full census. The result is a selective, verifiable map—not exhaustive by design. We *exclude*, conversely: (a) methods that are nearly equivalent to chosen representatives; (b) systems that do not *vary* a candidate inside a loop — e.g. natural-language-to-model translators or one-shot optimization-modeling assistants, which are adjacent but out of scope; and (c) any method that could not be read against a primary source.

Applying the classification rule. For each method class below, we apply the classification rule from Section 3: examine the operative prompt content, identify the dominant channel, and audit the label by drop-channel ablation; where ablation is inconclusive, we follow the source-described design intent. We state what each method class is conditioned on, what it emits, and its essential mechanism; full one-line classifications can be found in Appendix B, compact pseudocode in Algorithm 3, method class positions on the lens in Fig. 7, and a broader coverage map in Appendix C. The comparison in Section 6 makes the trade-offs between different method classes explicit. Among the representative methods in Table 2, this yields three `Numeric`-dominant, four `Symbolic`-dominant, and five `Linguistic`-dominant methods; the broader coverage map shows the same rapid, recent expansion, with most methods appearing during 2023–2026.

Discrete vs. continuous problems. One structural distinction cuts across the method classes: the type of the considered optimization problem shapes *where* the operator can act. In *discrete* problems the candidate is already a structured combinatorial representation, so the operator can receive that representation and emit another solution or a heuristic directly, and feasibility is a structural check ("each city once"); most automatic heuristic design work lives here (TSP, bin packing, scheduling). In *continuous* problems—where solutions are real-valued vectors—LLMs propose precise numbers

Algorithm 3 Skeletons of three code-emitting automatic heuristic design methods.

LLaMEA — (1+1) algorithm mutation [97]

```
best ← an initial algorithm
while budget remains do
    r ← (best, f(best), validation error or performance feedback)
    out ← SampleLLM(r) {mutates the current best algorithm}
    validate and score out; if f(out) is at least as good then best ← out
end while
return best
```

FunSearch — island program search [82]

```
initialize islands of programs, each scored by an evaluator f
while budget remains do
    pick an island; sample r from its high-scoring programs (exemplars)
    out ← SampleLLM(r) {proposes a new program conditioned on code exemplars}
    validate out; if valid (parses, matches interface, executes) then insert out scored by f
    trim lowest-scoring programs when the island exceeds capacity
end while
return best program
```

EoH — co-evolved idea + code [59]

```
initialize a population of (idea, code) pairs
while budget remains do
    select parents; assemble r from their (idea, code) pairs via explore or modify strategy
    out ← SampleLLM(r) {emits a new natural-language idea paired with code}
    score out by f; retain population's best
end while
return best
```

poorly at scale and have no native gradient. The usual design response is to place the LLM at the algorithm-design level: it emits an *algorithm* or operator, and a classical numerical routine handles the fine-grained search over vectors. Accordingly, continuous work — such LLaMEA applied to the BBOB black-box suite [97] and FunBO's discovered acquisition functions for Bayesian optimization [3]—is dominated by symbolic-conditioned, amortized operators: the LLM emits the search machinery offline; a classical method runs it. The lens organizes this split: discrete problems support operators at every conditioning level, including direct solution edits, whereas continuous problems tend to shift the operator up to the symbolic (algorithm-design) level.

5.1 `Numeric`-conditioned methods

Most `Numeric`-conditioned methods in this survey are transient, solution-level operators that use LLM calls to vary encoded candidates inside the search loop; *OPRO* and *EvoPrompt* are text-artifact boundary cases discussed below. *LMX* is closest to crossover: the prompt presents parent encodings, such as bit strings or sequences, and the LLM returns an offspring encoding [74]; *LLMOA* is a nearby hyper-heuristic case in which Gemini acts as a high-level component that constructs optimization sequences over low-level heuristics [124]; and *EvoLLM* keeps an evolutionary scaffold while using fitness-ranked solution vectors to steer the next update [54]. *LMEA* is another in-loop evolutionary case: in its TSP demonstration, parent tours and tour lengths condition LLM crossover and mutation over candidate tours [65]. The strength of these methods lies in their breadth: the prompt does not require executable machinery or a verbal rationale. The trade-of is that constraints, locality, and representation-specific structure usually have to be enforced by the encoding, move set, or validator rather than expressed explicitly in the prompt.

OPRO and *EvoPrompt* are interesting boundary cases because they optimize *prompts* scored on reasoning benchmarks rather than solutions to classical optimization problems. They are included here because the load-bearing conditioning signal is numeric: scored candidates, trajectories, or parent fitness values drive the search, even though the artifact being emitted and varied is text rather than a numeric solution encoding. *OPRO* prompts the LLM with previously generated candidates and their scores under a fixed task instruction [109]. *EvoPrompt* runs a genetic-algorithm/differential-evolution (GA/DE) skeleton in which the individuals are prompts, mutation/crossover are LLM calls, and scores guide parent selection [37]. A variant driven mainly by search-derived critiques or explanations would instead be `Linguistic`; a dagger († symbol, see Fig. 7) marks this dependence on the prompt design.

5.2 `Symbolic`-conditioned methods

Where `Numeric` methods primarily condition proposals on evaluative signals, `Symbolic` methods condition them on formal artifacts that can be parsed, executed, or checked by an external tool. Most `Symbolic` methods in the current combinatorial optimization landscape are code-emitting automatic heuristic design systems.[10] Here the operator searches *program space* rather than solution space: the LLM is conditioned on code and emits code—a heuristic, an operator, or a whole algorithm—that can be deployed and run without the LLM. Two preconditions define this method type. A *fast automatic evaluator* must exist, because every candidate program is scored by executing it; and the emitted artifact must be executable and inspectable, which is what makes the cost `Amortized`—the LLM is paid once, offline, and the emitted program then runs at no LLM cost.

The members differ mainly in how they organize the search. *FunSearch* evolves programs in island populations scored by an evaluator, maintaining high-scoring programs per island [82]; *AEL* is an early automatic-heuristic-design loop that evolves heuristic code with an LLM generator and an execution evaluator [60]; *LLaMEA* drives a (1+1) evolutionary loop over algorithm code [97]; and *LLM-LNS* evolves neighborhood-selection heuristics for large-neighborhood search in MILP while a second prompt-strategy layer revises how those heuristics are generated [114]. Further systems apply the same offline code-improvement pattern to whole optimizers and to non-expert workflows [16, 17]. We classify LLM-LNS as `Amortized` because the evolved neighborhood-selection heuristic is the artifact reused by the LNS solver, even though the underlying LNS procedure still operates inside a solve loop.

A second group of `Symbolic` methods keeps the offline generate–evaluate–retain pattern but changes how proposals are organized: *MCTS-AHD* organizes proposals through tree search [123], *FunBO* searches over code that defines Bayesian-optimization acquisition functions [3], and *AlphaEvolve* uses an agentic code-search loop with automated evaluators and a program database to evolve larger algorithmic codebases [77]. Unlike single-function heuristic search, it proposes and tests edits across a codebase, stores evaluated programs, and uses automated feedback to filter subsequent proposals.[11]

`Symbolic` conditioning is not limited to program code. Non-code `Symbolic` artifacts appear when the LLM varies a formal representation that can be parsed and checked by external tooling. For example, *MOLLEO* uses LLM mutation and crossover over SMILES molecular strings inside an evolutionary loop [99], while *LLEMA* proposes structured crystallographic specifications for multi-objective materials discovery, which are then checked and scored by materials-domain

[10] Code-emitting does not by itself imply `Symbolic` dominance: *EoH*, *ReEvo*, and *MEoH* emit code but are classified as `Linguistic`-dominant by the dominant-channel rule (Section 3), because the decisive conditioning signal is a natural-language idea (*EoH* and *MEoH*) or reflection (*ReEvo*) that steers code edits. We classify them as `Linguistic`-conditioned methods below.

[11] *LLM4AD* provides a shared platform for several automatic-heuristic-design methods in this family [62].

evaluators [2]. GI-LLM is the code-level counterpart: the LLM acts as a mutation operator inside genetic improvement, emitting source-code patches that are parsed and tested in the loop [12]. These cases also show that `Symbolic` conditioning need not be `Amortized`: when the formal artifact itself is the in-loop candidate, the persistence descriptor is `Transient`.

Algorithm 3 is an abstraction of the shared generate–evaluate–retain loop. *EoH* is included as a contrast case: although it emits code, the prompt signal steering each edit is `Linguistic`, so emitted artifact type alone does not determine the dominant conditioning channel.

5.3 `Linguistic`-conditioned methods

Here the operative signal is search-derived natural language (NL), typically a reflection on previous generations, a critique of a candidate, or an accumulated design principle. Because this text can appear at different levels of abstraction, three persistence settings are useful to distinguish: language-guided search (`Transient`), language-steered code generation (`Amortized`), and principle accumulation (`Transfer`). Language-guided search is the `Transient` form: *Self-Guiding Exploration (SGE)* uses thought trajectories generated during solving to decompose, execute, and refine combinatorial problem instances, while *MAEF* uses role-specialized agents whose reasoning and evaluator feedback steer schedule generation. For *MAEF*, the role-agent reasoning is the load-bearing linguistic signal; the evaluator scores are supporting feedback rather than the dominant channel. Neither method emits a reusable artifact [48, 100]. Language-steered code generation is usually `Amortized`: it emits executable code, but the edits are steered by NL ideas or critiques inside the offline design loop. *EoH* co-evolves an idea with its code implementation; *ReEvo* is the explicitly reflection-based case, inserting a reflection step between generator calls so short- and long-term critiques condition later code proposals; and *MEoH* extends the idea–code pattern with multi-objective selection over heuristic candidates [59, 111, 113]. Portfolio and diversity extensions refine that pattern: *EoH-S* emits a set of complementary heuristics, while *HSEvo* adds harmony-search and genetic-algorithm diversity machinery to keep heuristic directions separated [58, 78]. The reflection inside an offline method such as *ReEvo* is `Transient` within the meta-search, whereas the final emitted code is the `Amortized` artifact.

Principle accumulation makes this rationale the persistent artifact. This is the route by which `Linguistic` methods most naturally approach `Transfer`: a verbalized rule can be inspected, edited, and re-bound to a new instance family or domain instead of rediscovered from scratch. Two representative `Transfer`-oriented examples are *LAPT* and *HiFo-Prompt*: *LAPT* distills successful neural-architecture-search runs into verbalized design principles, while *HiFo-Prompt* combines hindsight principles with a foresight module that monitors population dynamics such as stagnation and diversity collapse to choose exploration or exploitation prompts [21, 125]. Other `Transfer`-labeled examples include *EvoPH*, which transfers co-evolved prompts and heuristic code; *MTHS*, which studies hierarchical cross-task heuristic design; and *EvolCAF*, which carries *EoH*-style design ideas into cost-aware acquisition-function design [63, 66, 112]. Their success depends on inspectability and re-bindability, and remains a claim to validate on the target task.

Across these settings, what NL adds over code is *rationale*: it can record *why* a change should help, preserve a design rule that can later be re-bound to a new instance family, or name a search direction that code and scores do not carry as directly. Such rationale can broaden the search over design ideas and help maintain diversity, but it does not make LLM-produced candidates correct by itself. The cost is verifiability: NL is the least checkable conditioning, so a wrong reflection can mislead the search, and the *operative-not-boilerplate* test (see Section 3) matters most here.

5.4 Boundary and hybrids

The cases described above assume that the LLM is the component proposing the next artifact. Several systems sit at the edge of that assumption. Memetic hybrids pair an LLM proposal with classical local search or repair [85]; controller-style methods use the LLM to route or schedule operators, connecting to adaptive operator selection and hyper-heuristics [13, 30, 83, 96]; and agentic solvers combine proposal, tool use, and orchestration during solving [110].

For these cases, the classification rule of Section 3 remains unchanged: we classify the method by the prompt signal used to propose the varied artifact, and annotate controller decisions, local-search post-processing, or tool orchestration separately. This keeps the catalogue from treating every auxiliary signal as a new dominant channel.

5.5 Classification overview

Fig. 7 shows classifications (mappings) of representative methods mentioned above with respect to the conditioning and persistence descriptors of Section 3 (the lens). Read each marker as a coordinate pair: the horizontal position records the dominant content type of the prompt, while the vertical position records what survives the call. The conditioning axis carries no ranking: `Symbolic` and `Linguistic` expose different kinds of structure and are incomparable taken as a whole. The persistence axis records increasing generalization scope, but not method quality. In the specific dimension of representation-portability, however, the three content types suggest a qualitative gradient from evaluative through denotational to propositional content: evaluative signals are closely tied to the search state; denotational signals are tied to a representation and can often be verified by execution; propositional signals can travel furthest but are less mechanically checkable. No channel dominates the other.

The current literature nevertheless maps onto the lower-right part of the layout: the chosen representative methods are classified on or below the diagonal formed by `Numeric/Transient`, `Symbolic/Amortized`, and `Linguistic/Transfer`. The diagonal is a layout choice, not an ordering of the conditioning axis; rearranging the unordered columns would produce a different visual pattern without changing the underlying tendencies. The tendency has a structural explanation: the persistence axis measures increasing generalization scope—from one iteration, to an entire run within a domain, to new domains—and the conditioning channel constrains how far an artifact can generalize. Evaluative content is specific to the search state that produced it; denotational content is specific to the representation and evaluator it was built for; propositional content is the most re-bindable because a later LLM call can re-interpret a design principle in a new context. Higher persistence therefore tends to require more domain-independent conditioning—not as a law, but as a structural tendency visible in the current body of methods. The methods below the diagonal prevent that pattern from becoming a rule: *MOLLEO*, *LLEMA*, and *GI-LLM* occupy the `Symbolic/Transient` position, while *SGE* and *MAEF* occupy `Linguistic/Transient` positions, so the diagonal describes where method families cluster, not where individual methods must fall.

The empty cells should therefore be read as gaps in the current research landscape, not impossibility claims. `Numeric/Amortized` and `Numeric/Transfer` methods have not yet been developed, but classical cross-domain hyper-heuristics show that score-based transfer is possible in non-LLM settings [13]. `Symbolic/Transfer` does not yet have a representative because denotational content typically names the objects and operations of the domain in which it was built; crossing domains can change those referents, requiring the artifact to be adapted rather than simply reused. Section 8 treats such missing conditioning–persistence pairings as empirical research questions.

6 Choosing an appropriate method: an evidence-based comparison

After the classification exercise of the previous section, we now turn from description to choice: which family should a practitioner use under a particular evaluator, budget, and deployment constraint?

Relevant dimensions. Important factors when deciding on one of the outlined methodologies are the intended application, the available budget, and runtime setting. Five dimensions govern that choice, developed here through an evidence table (Table 2) and a rule-based decision guide (Fig. 8); based on the descriptors of Section 3 and the build choices of Section 4. The first is *call-locus* (equivalently, cost-locus): whether the model is invoked during offline design or inside the search loop at each move or iteration. This choice matters because it changes where the model inference cost is paid and whether the deployed system remains model-dependent.[12] The second is *solution quality*, assessed here as source-specific evidence rather than a cross-method ranking: methods often serve different purposes, and even similar methods usually report on their own benchmarks and evaluators (Table 2). The evidence therefore supports comparison against each source's own baselines more than comparison across rows [6]. Because a shared performance matrix does not yet exist, the decision guide is rule-based: it asks practical questions about evaluator readiness, budget, call-locus, and persistence rather than ranking methods statistically. The third is *LLM cost and latency*: this is the dimension classical metaheuristics usually do not have to account for. Invalid output and bounded repair can inflate the per-accepted-candidate cost substantially. Offline methods pay this cost during meta-search but may deploy a reusable artifact without model calls; in-loop methods pay during the run, often per iteration. Section 7 gives the full accounting list. The fourth is *robustness and reproducibility*, which determine how much confidence to place in any single-run headline: stochastic decoding, prompt sensitivity, model/version drift, and evaluator implementation choices can all change the observed result. The fifth is *artifact persistence*: whether the emitted artifact can be reused after generation without keeping the model available at serving time, or whether the optimization loop remains a permanent model-dependent system. The guide below routes mainly on evaluator readiness, cost, call-locus, and persistence; solution quality and reproducibility act as verification filters through the evidence table and the shared reporting controls.

An evaluative comparison. With the dimensions identified, Table 2 evaluates representative methods on conditioning, emitted artifact, and call-locus, and reports each method's *own* headline result. These entries are source-specific evidence, not a *leaderboard*: the methods are evaluated on different benchmarks—extremal combinatorics, the BBOB black-box suite, online bin packing, and prompt-optimization tasks—and their headline numbers do not support direct cross-method ranking. The *Evidence* column therefore marks whether a result is peer-reviewed or preprint evidence, and whether it is source-local, matched, or qualitative; source-specific and preprint rows are useful signals, but not settled comparative evidence. Matched evidence is available only in narrow AHD groups. In online bin packing, *MCTS-AHD*, *FunSearch*, *EoH*, and *ReEvo* are scored under the same evaluator: *MCTS-AHD* achieves the smallest reported BPP gap (0.89%), and *EoH* reports a 20-generation online-BPP run (2,000 LLM queries) that matches or improves *FunSearch* in the paper's own evaluation. *HSEvo* provides a second matched comparison over BPO, TSP, and OP, emphasizing the diversity–quality trade-off. Even in these groups, the comparison is not a channel ablation: channel and search mechanism co-vary across the methods, and *MCTS-AHD*'s

[12] This is the closest analogue to algorithm selection [51, 53, 81]: choose a method based on the features and constraints of the problem instance. Standard algorithm selection usually requires a shared performance matrix over methods, instances, and budgets; such a matrix does not yet exist for LLM variation operators.

Table 2. Evidence summary for representative LLM variation operators. Column **Conditioning** indicates the dominant conditioning channel. *Call locus*: per op. = per operator application, per gen. = per generation, per step = per search step. *Evidence*: PR = peer-reviewed, pre = preprint, own = source-specific benchmark, matched = same evaluator, qual = qualitative only. Rows marked *pre* or *own* require the corresponding caution when read outside their source setting. The † symbol marks boundary cases.

Method	Conditioning	Emitted	Call locus	Evidence	Reported result
LMX [74]	`Numeric`	solution	per op.	PR; qual	demonstrates semantic crossover behavior on synthetic tasks; no headline metric on classical optimization benchmarks
OPRO [109]	`Numeric`†	prompt / solution	per step	PR; own	GSM8K up to +8 p.p.; some BBH tasks up to +50 p.p. vs. human prompts
EvoPrompt [37]	`Numeric`†	prompt	per gen.	PR; own	some BBH tasks up to +25 p.p. vs. human & auto prompts
FunSearch [82]	`Symbolic`	program	offline	PR; own	cap set n=8: 512, bound $C \geq 2.2202$; beats best-fit on online BPP
LLaMEA [97]	`Symbolic`	algo. code	offline	PR; own	BBOB subset at 5D: outperforms CMA-ES & DE on several functions under reported budget
MCTS-AHD [123]	`Symbolic`	program	offline	PR; matched	smallest matched online-BPP gap: 0.89% vs. FunSearch/EoH/ReEvo
AlphaEvolve [77]	`Symbolic`	program / code	offline	pre; own	rank-48 tensor decomposition for 4×4 complex matmul; first characteristic-0 improvement over Strassen's rank-49 and 14 matrix targets improved
EoH [59]	`Linguistic`	idea + code	offline	PR; own	online BPP: 20 generations / 2,000 LLM queries; matches or improves FunSearch in its own evaluation
ReEvo [113]	`Linguistic`	code	offline	PR; own	competitive on 6 CO problems, more sample-efficient
HSEvo [78]	`Linguistic`	code population	offline	PR; matched	BPO/TSP/OP matched study: improves or matches FunSearch/EoH/ReEvo while tracking diversity
LAPT [125]	`Linguistic`	principles	offline	PR; own	beats prior transferable-NAS (TNAS) methods on most tasks
MEoH [111]	`Linguistic`	pareto set of heuristics	offline	PR; own	up to 10× more efficient; more trade-offs

advantage may reflect its tree-search structure rather than its `Symbolic` conditioning channel. The drop-channel ablation discipline of Section 7 is designed to partially separate such effects. The table is representative rather than exhaustive; some methods discussed in the survey are not assigned separate evidence rows.

Fair comparison requirements. A reported result should guide method choice only when the experimental controls support the particular comparison being made. At least, a credible within-study comparison should use the same evaluator and harness, compare against a strong classical baseline where available, include a no-LLM or drop-channel ablation, and match the relevant budgeted resources; Section 7 gives the full reporting list, including repair attempts and accepted candidates. Preprint and source-specific rows in Table 2 remain informative, but should be interpreted within their original experimental setting until comparable controls are available. Across benchmark sets, best-run claims should be replaced by repeated runs and appropriate statistical tests [6, 26]. Section 7 discusses the failure modes these controls are intended to reveal.

A decision guide. The first branch is the null choice: add an LLM variation operator only when it adds a capability that a classical operator, hyper-heuristic, or algorithm-configuration method does not. The operative test is a capability gap, not model availability. Prefer a classical or non-LLM design when, for example, the move family for your local search is already well explored and tested, when inference or evaluation cost cannot be justified at the required search budget, or when model outputs cannot be automatically parsed, validated, and evaluated; in each case, the added LLM layer introduces cost and failure modes without a clear compensating role. An LLM earns its place when it can explore structured artifact spaces that are too large to enumerate manually, extract search guidance from heterogeneous signals—objective values, constraint violations, and natural-language annotations—that resist encoding as fixed operator logic, or define operator move families in domains where the appropriate rules are not known in advance. In all cases, measure the gain against a matched no-LLM baseline under the same evaluation budget; if the advantage is not retained under that control, additional search effort may explain the result. The no-LLM baseline therefore serves two roles: it is both an evaluation control and the first design gate.

Figure 8 turns this into a staged flowchart; Table 2 is its evidence companion. The *Call locus* column maps to the locus gate, the *Emitted* column names the deployable or runtime artifact, and the *Evidence* and *Reported result* columns state how source-specific each result is. The flowchart operationalizes call-locus and LLM cost as branching decisions; solution quality and reproducibility reappear in the verification step at the bottom. Three gate questions route the practitioner:

(1) Is an automated evaluation pipeline—including evaluator, validator, and repair mechanism—in place? This is a prerequisite for any iterative LLM-based search: without it, generated artifacts cannot be reliably checked, scored, or selected (Section 4, Algorithms 1–2).
(2) Can many candidates be evaluated? This scale depends on the number of candidates considered per generation. Offline automatic heuristic design (in the style of *FunSearch*, *EoH*, *LLaMEA*, or *MCTS-AHD*) can place substantial pressure on evaluator throughput because meta-search repeatedly evaluates generated artifacts [59, 82, 97, 123]. When throughput is limited, a query-efficient loop, careful evaluation budgeting, or surrogate/acquisition-based screening is required [3].
(3) Is runtime LLM inference affordable within the wall-clock and token budget? Offline methods move model inference to a single design phase and can subsequently deploy a reusable artifact without further model calls, whereas in-loop methods incur model cost during optimization.

Once these questions have been answered, the remaining choices identify the operator family. On the offline path, the key question is whether natural-language ideas or reflections should be explicit design objects, or whether the population should contain only formal code artifacts. On the in-loop path, the key question is whether the LLM directly modifies candidate artifacts or supplies guidance that a conventional operator executes; if it modifies candidates directly, the candidate's formal verifiable structure further separates formal-artifact methods from numeric solution variation. These are structural distinctions: they follow from what the LLM emits and what the downstream pipeline consumes. The dashed refinements under Language-steered methods mark two special cases: principle accumulation is preferred when a domain-invariant principle can be identified and validated across problem types, whereas diversity-seeking variants are warranted when the proposal distribution collapses toward too-narrow a family of candidates (Section 7). The binary branching is a readability simplification; in practice, hybrid schedules can call the model every k iterations, only at restarts, or only when the search stagnates. Within the selected family, final method choice still requires target-task validation, budget parity, and drop-channel ablation under the fair-comparison requirements above; Table 2 supplies source evidence but does not replace these checks. The verification bar in Fig. 8 summarizes this handoff from family selection to evidence validation.

The guide should therefore be read as a routing aid for family architecture, not as a universal selector or a complete experimental claim. It narrows whether the appropriate starting point is an offline artifact, an in-loop solution-variation operator, a language-guided controller, a transfer-oriented principle system, or a portfolio/diversity variant. The branches are binary for readability, but real deployments couple these choices with evaluator readiness, validation and repair, evaluation-call budget, token budget, wall-clock constraints, and reproducibility requirements. Section 7 addresses these shared evaluation and engineering risks, which must be resolved before any result drawn from the guide can be trusted.

7 Cross-cutting concerns

The choice of the LLM-based variation operator type is only the first step. The resulting method still inherits engineering and evaluation risks shared across the lens. The concerns below are operational rather than taxonomic: they affect `Numeric`, `Symbolic`, and `Linguistic` operators, and they should be addressed before any reported improvement is interpreted as evidence.

Prompt sensitivity and ablation discipline. The conditioning channel is not the whole prompt. Wording, example order, delimiters, schema fields, and the position of load-bearing content can alter LLM proposals even when the channel class is unchanged [70, 75, 87, 121]. Because models condition on the exact preceding text, not on an abstract representation of what that text means, long histories create a placement risk: decision-critical information may be used less reliably when buried in the middle of the context window [64]. Critical conditioning should therefore remain close to the instruction, and any summarization or retrieval rule used to fit the context window should be treated as part of the prompt design and archived with the template. A credible study should release the exact template, test a small set of meaning-preserving variants, and distinguish formatting failures from search failures. Drop-channel ablation identifies the load-bearing signal, but it does not test sensitivity to wording, examples, or schemas within the same channel. When supported by the backend, structured-output APIs, tool/function calling, and grammar-constrained decoding can constrain generation and localize format failures [32, 102]; validation remains necessary, and schema names or field descriptions may themselves carry instructional signals. Substantive schema changes should therefore be frozen or ablated alongside prompt variants. Hallucination is a related grounding failure: plausible content unsupported by the specification, API, codebase, or evaluator

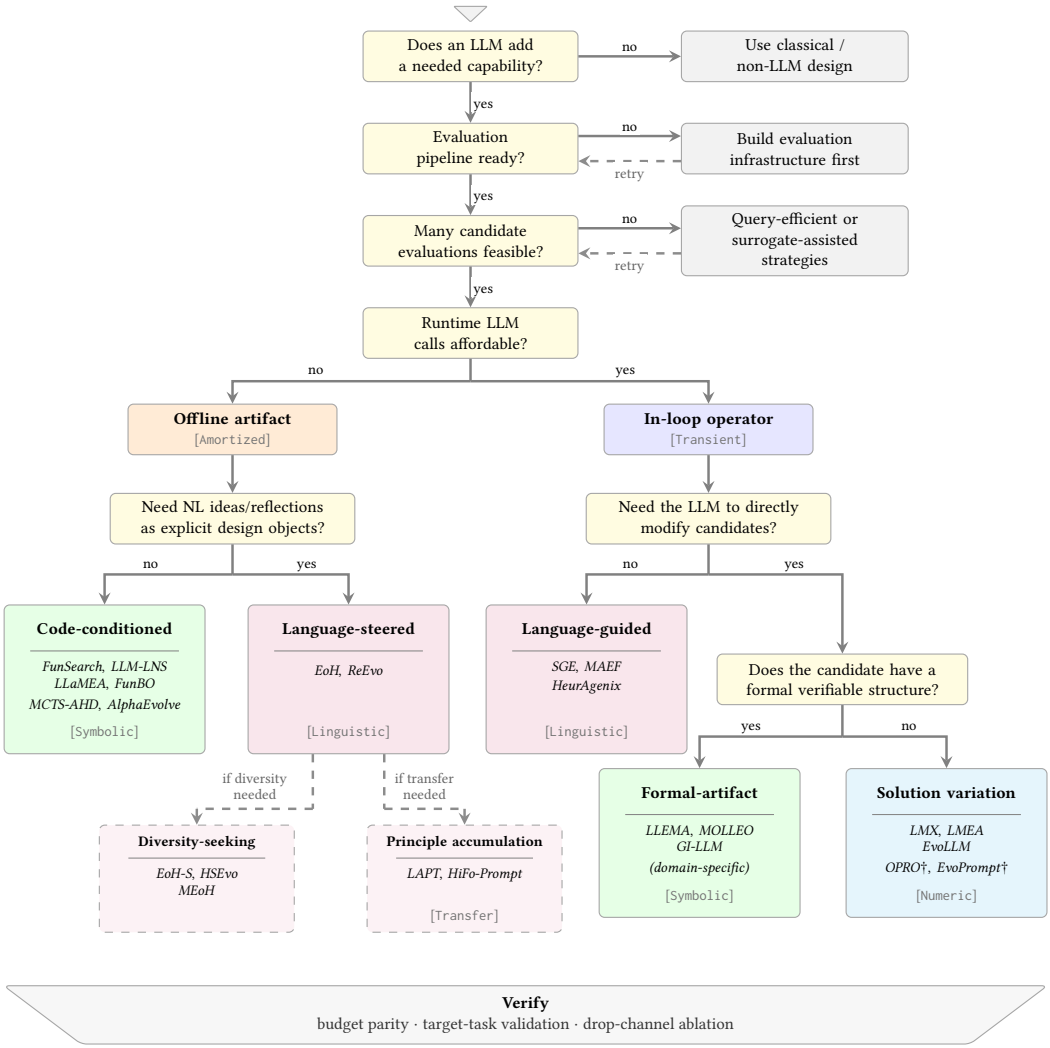

Fig. 8. Decision guide for choosing whether and how to use an LLM variation operator. The verification bar summarizes the required reporting checks.

can introduce false constraints, invalid calls, spurious diagnoses, or unsafe code paths into the search state. Generated content should pass the relevant schema, validator, sandbox, source check, or local evaluation before conditioning later prompts. Lower temperature or narrower top-p may reduce variance and format drift, but cannot prevent unsupported claims.

Diversity at the proposal step. Diversity collapse can occur at generation time: LLM priors from training, alignment, and decoding may make repeated calls produce similar candidates before acceptance or selection begins. This differs from evolutionary search, where diversity is typically reduced by selection, replacement, or local improvement; with an LLM operator, the proposed candidates may already be too similar before those mechanisms act. Downstream diversity mechanisms can therefore only act on the generated variants. Methods such as *FunSearch*, *EoH-S*, *HSEvo*,

and *MEoH* address this through islands, set-valued output, diversity pressure, or multi-objective search [58, 78, 82, 111]. Diversity can be measured by edit or syntactic distance, behavioral spread on probe instances, or objective-space coverage, and should be reported over time alongside objective value. Mitigation targets inputs (diverse examples or histories), populations (islands or distinct generators),[13] and decoding (higher temperature or top-p while preserving an acceptable valid-output rate [44]).

Evaluation validity. Budget comparisons are often ambiguous: an "evaluation" may denote an objective call, an LLM sample, a token budget, a repaired/accepted candidate, or wall-clock time. These quantities are not directly comparable and should therefore be reported separately: evaluator calls, model calls/samples, input/output tokens, repair attempts, accepted candidates, and wall-clock.[14] Budget parity is still insufficient if the scoring source changes: LLM judges can conflate search quality with evaluator preferences, especially when the generator and judge share a model family [89, 122].[15] Thus, comparisons require identical evaluators, strong baselines, no-LLM or drop-channel controls, repeated runs, and appropriate statistical tests [6, 26]. Contamination by pre-training data is a related community-level benchmarking issue, treated in Section 8; a practical guard is to run a zero-shot or held-out probe before treating search success on a public benchmark as evidence of generalization.

Latency, batching, and infrastructure. Embedding an LLM in the variation operator changes runtime shape: unlike local-search moves, LLM calls are remote, high-variance, and often latency- or quota-bound. Population methods can hide part of this latency through batching or asynchronous calls. Self-hosting shifts bottlenecks to local hardware and serving configuration, whereas API access adds rate limits, provider-side latency variance, prompt-logging/data-retention constraints, and possible moderation or safety filters; these issues matter especially for sensitive optimization data. Batching and asynchronicity help only when generated artifacts can pass automatically through parsing, feasibility checks, sandboxing, bounded repair, and diagnostics — the infrastructure required by the first gate in Fig. 8. For code-emitting or tool-using operators, execution safety belongs in the evaluation pipeline. In new domains, this infrastructure may be the main cost of entry.

Operator non-stationarity. LLM variation operators are time-dependent in a way most classical operators are not: model updates, endpoint changes, or decoding defaults can alter the induced proposal distribution, creating a moving-baseline problem [22]. Claims therefore depend on the model version, endpoint/checkpoint date, decoding settings, and repair policy used to instantiate the operator. This is especially important for `Transient` operators, which remain model-dependent at every run; thus, reproduction across model families or pinned checkpoints strengthens the evidence. For offline-`Amortized` methods, the model belongs to the design process rather than the deployed product. Releasing the generated heuristic, program, or operator, together with model, prompt, log, evaluator, seed, and environment details, is therefore central to reproducibility [34, 67, 79].

Taken together, these concerns define the minimum requirements for credible LLM-in-metaheuristics studies: prompt and history handling, diversity reporting, matched evaluation controls,

[13] Distinct generators may use different prompts, model checkpoints, or even different models across islands.

[14] `Transient` operators pay per use, amortized methods shift cost to meta-search, and batching or API latency can change runtime independently of the algorithm. For self-hosted inference, accelerator time and measured energy should be reported when available; for API models, calls and tokens are imperfect proxies for monetary and infrastructure cost [42, 91].

[15] Position bias, style preference, and self-consistency effects can create circular evidence rather than objective improvement.

safe execution, and artifact release must be specified before a claimed improvement can be interpreted as operator evidence. Section 8 asks what the field still lacks even after these conditions are met.

8 Open problems and research directions

Section 7 identified recurring risks, mitigations, and reporting controls needed for credible LLM-in-metaheuristics studies. The directions below go beyond these baseline requirements: they remain open because the field still lacks the theoretical foundations, infrastructure, and empirical evidence needed to address them fully and reliably.

Theoretical foundations. The proposal-distribution notation $q(x' \mid r)$ makes design effects observable, but it is not yet a theory of LLM-induced search. We do not know which proposal properties — support over feasible artifacts, locality, diversity, or probability mass near high-quality candidates — would support finite-budget guarantees, nor how those properties depend on the conditioning channel in r. The natural object is the stochastic process induced by proposal, validation, repair, and acceptance — a Markov chain when the search state is explicit. This separates asymptotic convergence from finite-budget performance, while allowing the effective proposal kernel to depend on prompt history, decoding, and validation or repair state. Classical theory gives useful anchors but also shows why broad claims would be premature: No-Free-Lunch results require assumptions on the problem class [104], and convergence results for Simulated Annealing or Genetic Algorithms assume transition kernels, schedules, or variation mechanisms that LLM operators do not automatically satisfy [40, 84]. EDA and Bayesian views of in-context learning offer analogies, not guarantees [55, 106]. Similarly, GSGP provides a behavioral foundation for program variation without covering prompt-conditioned edits, natural-language rationales, or model priors [76, 98]. The theory of LLMs and in-context learning is itself still partial, so a general optimization theory built on LLM-induced proposal kernels remains premature. Progress should therefore start with restricted claims: finite-budget results for validated LLM proposal families, or measurable conditions under which a channel increases valid-proposal mass or quality without collapsing diversity.

Benchmarking infrastructure. LLM-based metaheuristics need benchmark infrastructure tailored to prompt-conditioned operators, since classical test sets may not expose evaluator differences, unequal token or repair budgets, contamination through known instances or solver traces, and sensitivity to artifact-validation harnesses. Thus, a fair classical benchmark may still fail to isolate the LLM operator's contribution. Existing efforts provide partial components: contamination studies motivate optimization-specific audits [107]; *LLM4AD* offers reusable generation/evaluation infrastructure [62]; *COCO* exemplifies controlled optimization benchmarking [41]; and the online bin-packing results in Table 2 demonstrate same-evaluator comparison across LLM-based automatic-heuristic-design methods. The open problem is to integrate these practices across domains [6]. A practical suite should include a parametric fresh-instance generator, shared evaluators, classical/no-LLM baselines, standardized accounting of evaluator/model calls, tokens, repairs, and time, multi-run statistical protocols [26], and contamination audits. Zero-/one-shot tests on hidden instances can reveal when search adds little beyond direct model capability, though they do not detect subtler benchmark familiarity. Robustness should also be tested under randomized, shifted, and adversarial instances, while transfer claims require cross-domain benchmarks with controlled source–target structural distance.

The cost-quality frontier. The question from Section 1 — whether more explicit conditioning improves performance — remains unresolved, with evidence suggesting task-dependent effects rather than a general rule. The key open direction is the cost-quality Pareto frontier: how solution or

heuristic quality trades off against token budget, repair rate, valid-output rate, energy or accelerator usage, and deployment portability, with model scale as a study variable. A central hypothesis is that representation design and model scale can partly compensate for one another: schemas, executable structure, and sharper diagnostics can reduce model burden, whereas loose or verbose conditioning increases token and context demands.

Testing this requires matched factorial comparisons: fix the evaluator, outer loop, and total resources; vary conditioning channel, call-locus/persistence, and model scale; and report quality per token, valid-output rate, repair rate, diversity, latency, and energy or accelerator usage. Although model scale remains confounded with architecture, training data, alignment, and serving stack [43, 49], matched comparisons can help separate representation effects from model-strength effects. *EoH*, *MEoH*, and *LLaMEA* provide partial evidence on quality and model-call costs [59, 97, 111]. A first credible frontier study should test whether channel choice shifts the quality-per-token curve under matched budgets and at least two model scales.

Auditability and interpretability. Auditability asks whether conditioning signals actually shape search. The drop-channel rule of Section 3 provides channel-level attribution, while content-level ablations test which history segments matter within a channel. Removing failures, successes, or reflections while preserving the prompt format establishes counterfactual sensitivity [72]: it shows that a prompt component changes the output, not how the model computes that change. Artifact-level audits complement prompt ablations. Edit distance, AST similarity, and clustering can reveal recurring heuristic families, while linguistic principles can be tested behaviorally by comparing artifacts generated with and without them. Such analyses show whether prompt changes alter the structural or behavioral family of generated artifacts, not only the final score. Recent work shows that LLM representations of combinatorial problem structure can be recovered by probing [24], but whether these representations actively shape $q(x' \mid r)$ during generation or are only recoverable after the fact remains unknown.

Interpretability asks the harder model-level question: how a large learned model turns the conditioning signal into a proposal. Chain-of-thought can expose stated reasoning [101], but it remains an output artifact rather than direct access to the internal computation that produced it. Circuit analysis and causal editing offer tools for identifying internal features and pathways [28, 73], yet their relevance to optimization artifacts remains open. Progress would require paired content ablations and artifact analyses showing that removing history components changes both performance and the structural or behavioral family of proposals. In agentic systems, attribution may instead target roles or tool calls.

Transfer and artifact portability. Transfer is the flagship persistence challenge: the goal is not merely more conditioning, but persisted artifacts that are inspectable and sufficiently instance- or domain-invariant to be re-bound without rediscovery. The sparse `Transfer` coverage in Fig. 7 and Table 4 reflects that most systems demonstrate reuse within a task distribution rather than out-of-distribution transfer. Three forms are relevant. First, executable transfer carries symbolic artifacts such as heuristic code, which must be sufficiently decoupled from the source evaluator to adapt to a target domain; *GEAKG* illustrates this direction [19]. Second, principle-level transfer carries declarative design rules through later prompts, as in *LAPT*, *HiFo-Prompt*, and *EvoPH*; *EvolCAF* similarly transfers an evolved acquisition-function design [21, 66, 112, 125]. Such principles may travel more easily than source-specific programs, but cannot be mechanically verified in the target domain. Third, meta-level transfer asks whether process-level knowledge for generating or selecting operators can bias or initialize search on new problem classes, echoing classical hyper-heuristics [13]; *MTHS* provides an early LLM-based example [63]. Progress would require controlled

source–target tests showing that persisted artifacts reduce target-domain design effort relative to rediscovery, rather than merely demonstrating reuse after manual adaptation.

Agentic and multi-agent metaheuristics. Agentic systems turn a single LLM operator into a role-structured process involving generators, critics, verifiers, planners, tool users, or orchestrators. Multi-agent systems are the case where multiple communicating model instances or separately role-prompted calls interact [38]. Among surveyed methods, *ReEvo* uses a generator–reflector split [113], while *MAEF* and *HeurAgenix* use richer role structures [100, 110]; *AlphaEvolve* instead combines generation, a program database, and automated evaluation [77].

Key open questions concern architecture and evaluation: which role compositions improve search, preserve diversity, and remain auditable at acceptable cost? Same-family critics may reproduce generator blind spots, embedding the LLM-as-judge circularity of Section 7 in the architecture. Role messages can propagate erroneous diagnoses, while diversity may collapse through generators, critics, or shared search vocabulary. Additional roles can multiply calls and repairs, and multi-role call graphs make failures harder to attribute [5]. Studies should therefore log call graphs, role-specific token costs, and accepted-artifact lineage. A convincing next step is to compare single- and multi-role systems under identical evaluators, budgets, and logging, isolating gains from role structure rather than extra calls, selection pressure, or blind-spot amplification.

Beyond prompt channels. Two frontiers extend beyond text-only prompt channels. Weight-internalized variation stores search signals in fine-tuned, adapter-based, or RL-trained parameters rather than inspectable prompts, as in *EvoTune* [93]; related neural combinatorial optimization directly constructs or guides solutions [7, 52]. If conditioning resides only in weights, the drop-channel rule no longer applies, and domain-specific operators raise the same cross-domain transfer question: can they generalize without retraining?

Multimodal conditioning uses non-text representations of instances or evolving solutions: *ViTSP* renders fixed problem instances, while *VEO* renders evolving incumbents or solution states [116, 120]. The key question is whether vision adds information beyond an equivalent textual description under identical loops and budgets. Hybrid text–visual prompts further ask whether visual input contributes complementary structure or merely duplicates information. Progress requires controlled comparisons of textual, visual, and weight-internalized conditioning, with the visual or weight-internalized signal removed or replaced as a drop-channel analogue.

9 Conclusion

This tutorial organizes LLM variation operators in metaheuristics through a two-descriptor framework: dominant prompt conditioning and artifact persistence. Conditioning asks what the prompt makes load-bearing at the variation step—numeric evidence, symbolic structure, or linguistic rationale. Persistence asks what survives the model call—a transient candidate used inside the loop, an amortized artifact reused after offline design, or a transfer artifact re-bound to a new family or domain. In the classical syntax/semantics sense, this is the paper's "semantic turn": variation moves from classical operators—which apply fixed, hand-coded moves, mutations, construction rules, or recombination rules to encoded states—to prompt-conditioned LLM operators in which the representation r is supplied in the prompt at generation time and becomes an explicit design choice. The content supplied through r can therefore be characterized as evaluative (Numeric), behavioural or denotational (Symbolic), or propositional (Linguistic). This does not claim that the model understands the problem or the representations it manipulates, nor that richer or more verbose prompts necessarily produce better search behaviour.

Operationally, the framework gives the tutorial three uses: method placement, operator construction, and family selection. Placement is based on the prompt content that is load-bearing at

generation time, not on the emitted artifact alone; the drop-channel audit makes that judgment reproducible. Construction treats the LLM call as one component in a complete variation loop: the prompt structure determines what information reaches the model; the completion format, parser, validator, and bounded-repair path determine which outputs become valid candidates; and the evaluator determines how accepted artifacts feed back into search. When choosing among alternatives, the null option remains the starting point: add an LLM only when it supplies a capability that classical operators, hyper-heuristics, or configuration methods cannot achieve. After that gate, the decision follows the practical constraints of the guide: whether the validation and evaluation infrastructure is ready, whether enough candidates can be evaluated, whether runtime calls fit the wall-clock and token budget, and whether the deployment setting favours offline artifact design or in-loop model use. Those answers determine the conditioning family the setting can support.

Reliable evidence requires more than correct operator placement. Prompt sensitivity, hallucinated or malformed outputs, diversity loss, evaluator mismatch, latency, infrastructure constraints, and model drift are risks that cut across all conditioning channels and method families. They call for a reporting discipline: generated artifacts should be validated, subject to bounded repair, and sandboxed when they execute code; comparisons should match evaluators, model budgets, and wall-clock limits; and studies should release prompts, artifacts, logs, and drop-channel or content-level audits. Several foundational gaps remain open: finite-budget theory for LLM-induced proposal distributions, contamination-resistant benchmark infrastructure, cost-quality evidence across conditioning richness and model scale, and transfer results that go beyond within-distribution reuse. Closing these gaps would make LLM-conditioned operators more than an expressive way to generate moves; it would make them a reproducible mechanism for extending metaheuristic search to problems where effective variation is difficult to hand-design.

Acknowledgments

Guillem Rodríguez-Corominas was supported by the European Commission–NextGenerationEU through the Momentum CSIC Programme: Develop Your Digital Talent, under project MMT24-IIIA-02.

References

[1] Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, et al. 2024. Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone. arXiv preprint arXiv:2404.14219.

[2] Nikhil Abhyankar, Sanchit Kabra, Saaketh Desai, and Chandan K. Reddy. 2026. LLEMA: Evolutionary Search with LLMs for Multi-Objective Materials Discovery. In *ICLR*.

[3] Virginia Aglietti, Ira Ktena, Jessica Schrouff, Eleni Sgouritsa, Francisco J. R. Ruiz, Alan Malek, Alexis Bellot, and Silvia Chiappa. 2025. FunBO: Discovering Acquisition Functions for Bayesian Optimization with FunSearch. In *ICML (PMLR, Vol. 267)*. 639–661.

[4] Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. 2024. OptiMUS: Scalable Optimization Modeling with (MI)LP Solvers and Large Language Models. In *ICML (PMLR, Vol. 235)*. 577–596.

[5] Adi Banerjee, Anirudh Nair, and Tarik Borogovac. 2025. Where Did It All Go Wrong? A Hierarchical Look into Multi-Agent Error Attribution. In *NeurIPS*.

[6] Thomas Bartz-Beielstein, Carola Doerr, Daan van den Berg, Jakob Bossek, Sowmya Chandrasekaran, Tome Eftimov, Andreas Fischbach, Pascal Kerschke, William La Cava, Manuel López-Ibáñez, Katherine M. Malan, Jason H. Moore, Boris Naujoks, Patryk Orzechowski, Vanessa Volz, Markus Wagner, and Thomas Weise. 2020. Benchmarking in Optimization: Best Practice and Open Issues. arXiv preprint arXiv:2007.03488.

[7] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. 2021. Machine Learning for Combinatorial Optimization: A Methodological Tour d'Horizon. *European Journal of Operational Research* 290, 2 (2021), 405–421. doi:10.1016/j.ejor.2020.07.063

[8] Christian Blum, Jakob Puchinger, Günther R. Raidl, and Andrea Roli. 2011. Hybrid metaheuristics in combinatorial optimization: A survey. *Applied Soft Computing* 11, 6 (2011), 4135–4151. doi:10.1016/j.asoc.2011.02.032

[9] Christian Blum and Andrea Roli. 2003. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *Comput. Surveys* 35, 3 (2003), 268–308. doi:10.1145/937503.937505

[10] Herbie Bradley, Andrew Dai, Hannah Teufel, Jenny Zhang, Koen Oostermeijer, Marco Bellagente, Jeff Clune, Kenneth Stanley, Grégory Schott, and Joel Lehman. 2024. Quality-Diversity through AI Feedback. In *ICLR*.

[11] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-Shot Learners. In *NeurIPS*.

[12] Alexander E. I. Brownlee, James Callan, Karine Even-Mendoza, Alina Geiger, Carol Hanna, Justyna Petke, Federica Sarro, and Dominik Sobania. 2023. Enhancing Genetic Improvement Mutations Using Large Language Models. In *SSBSE (LNCS, Vol. 14415)*. Springer, Cham, 153–159. doi:10.1007/978-3-031-48796-5_13

[13] Edmund K. Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. 2013. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society* 64, 12 (2013), 1695–1724. doi:10.1057/jors.2013.71

[14] Edmund K. Burke, Matthew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and John R. Woodward. 2019. A Classification of Hyper-Heuristic Approaches: Revisited. In *Handbook of Metaheuristics*. International Series in Operations Research & Management Science, Vol. 272. Springer, Cham, 453–477. doi:10.1007/978-3-319-91086-4_14

[15] Laura Calvet, Jésica de Armas, David Masip, and Angel A. Juan. 2017. Learnheuristics: hybridizing metaheuristics with machine learning for optimization with dynamic inputs. *Open Mathematics* 15, 1 (2017), 261–280. doi:10.1515/math-2017-0029

[16] Camilo Chacón Sartori and Christian Blum. 2025. Optimizing the Optimizer: An Example Showing the Power of LLM Code Generation. In *Proceedings of the 20th Conference on Computer Science and Intelligence Systems (FedCSIS) (Annals of Computer Science and Information Systems, Vol. 43)*. 47–57. doi:10.15439/2025F1481

[17] Camilo Chacón Sartori and Christian Blum. 2026. Combinatorial Optimization for All: Using LLMs to Aid Non-Experts in Improving Optimization Algorithms. *Inteligencia Artificial* 29, 77 (2026), 108–132. doi:10.4114/intartif.vol29iss77pp108-132

[18] Camilo Chacón Sartori, Christian Blum, Filippo Bistaffa, and Guillem Rodríguez Corominas. 2025. Metaheuristics and Large Language Models Join Forces: Toward an Integrated Optimization Approach. *IEEE Access* 13 (2025), 2058–2079. doi:10.1109/ACCESS.2024.3524176

[19] Camilo Chacón Sartori, José H. García, Andrei Voicu Tomut, and Christian Blum. 2026. GEAKG: Generative Executable Algorithm Knowledge Graphs. arXiv preprint arXiv:2603.27922.

[20] Angelica Chen, David M. Dohan, and David R. So. 2023. EvoPrompting: Language Models for Code-Level Neural Architecture Search. In *NeurIPS*, Vol. 36. 7787–7817. doi:10.52202/075280-0342

[21] Chentong Chen, Mengyuan Zhong, Ye Fan, Jialong Shi, and Jianyong Sun. 2026. HiFo-Prompt: Prompting with Hindsight and Foresight for LLM-based Automatic Heuristic Design. In *ICLR*.

[22] Lingjiao Chen, Matei Zaharia, and James Zou. 2024. How is ChatGPT's Behavior Changing over Time? *Harvard Data Science Review* 6, 2 (2024). doi:10.1162/99608f92.5317da47

[23] Artur Leandro da Costa Oliveira, André Britto, and Renê Gusmão. 2023. Machine learning enhancing metaheuristics: a systematic review. *Soft Computing* 27 (2023), 15971–15998. doi:10.1007/s00500-023-08886-3

[24] Francesca Da Ros, Luca Di Gaspero, and Kevin Roitero. 2025. Behavior and Representation in Open-Weight Large Language Models for Combinatorial Optimization: From Feature Extraction to Algorithm Selection. arXiv preprint arXiv:2512.13374; accepted for publication in Computers & Operations Research.

[25] Francesca Da Ros, Michael Soprano, Luca Di Gaspero, and Kevin Roitero. 2026. Large Language Models for Combinatorial Optimization: A Systematic Review. *Comput. Surveys* 58, 11 (2026), 1–53. doi:10.1145/3801961

[26] Joaquín Derrac, Salvador García, Daniel Molina, and Francisco Herrera. 2011. A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. *Swarm and Evolutionary Computation* 1, 1 (2011), 3–18. doi:10.1016/j.swevo.2011.02.002

[27] Marco Dorigo and Thomas Stützle. 2004. *Ant Colony Optimization*. MIT Press. doi:10.7551/mitpress/1290.001.0001

[28] Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. 2021. A Mathematical Framework for Transformer Circuits. *Transformer Circuits Thread* (2021). https://transformer-circuits.pub/2021/framework/index.html

[29] Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2024. Promptbreeder: Self-Referential Self-Improvement via Prompt Evolution. In *ICML*.

[30] Álvaro Fialho, Luis Da Costa, Marc Schoenauer, and Michèle Sebag. 2010. Analyzing bandit-based adaptive operator selection mechanisms. *Annals of Mathematics and Artificial Intelligence* 60, 1 (2010), 25–64. doi:10.1007/s10472-010-9213-y

[31] Michel Gendreau and Jean-Yves Potvin (Eds.). 2019. *Handbook of Metaheuristics* (3 ed.). Springer. doi:10.1007/978-3-319-91086-4

[32] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. 2023. Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning. In *EMNLP*. 10932–10952. doi:10.18653/v1/2023.emnlp-main.674

[33] Fred Glover. 1986. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research* 13, 5 (1986), 533–549. doi:10.1016/0305-0548(86)90048-1

[34] Odd Erik Gundersen and Sigbjørn Kjensmo. 2018. State of the Art: Reproducibility in Artificial Intelligence. In *AAAI*, Vol. 32. doi:10.1609/aaai.v32i1.11503

[35] Ping Guo, Fei Liu, Xi Lin, Qingchuan Zhao, and Qingfu Zhang. 2024. L-AutoDA: Leveraging Large Language Models for Automated Decision-based Adversarial Attacks. In *GECCO Companion*. 1846–1854. doi:10.1145/3638530.3664121

[36] Ping Guo, Qingfu Zhang, and Xi Lin. 2026. CoEvo: Continual Evolution of Symbolic Solutions Using Large Language Models. In *AAAI*, Vol. 40. 1810–1818.

[37] Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. 2024. Connecting Large Language Models with Evolutionary Algorithms Yields Powerful Prompt Optimizers. In *ICLR*.

[38] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large Language Model based Multi-Agents: A Survey of Progress and Challenges. In *IJCAI*. 8048–8057. doi:10.24963/ijcai.2024/890

[39] Kilem L. Gwet. 2008. Computing inter-rater reliability and its variance in the presence of high agreement. *Brit. J. Math. Statist. Psych.* 61, 1 (2008), 29–48. doi:10.1348/000711006X126600

[40] Bruce Hajek. 1988. Cooling Schedules for Optimal Annealing. *Mathematics of Operations Research* 13, 2 (1988), 311–329. doi:10.1287/moor.13.2.311

[41] Nikolaus Hansen, Anne Auger, Raymond Ros, Olaf Mersmann, Tea Tušar, and Dimo Brockhoff. 2021. COCO: A Platform for Comparing Continuous Optimizers in a Black-Box Setting. *Optimization Methods and Software* 36, 1 (2021), 114–144. doi:10.1080/10556788.2020.1808977

[42] Peter Henderson, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, and Joelle Pineau. 2020. Towards the Systematic Reporting of the Energy and Carbon Footprints of Machine Learning. *Journal of Machine Learning Research* 21, 248 (2020), 1–43.

[43] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, et al. 2022. Training Compute-Optimal Large Language Models. In *NeurIPS*, Vol. 35. 30016–30030. doi:10.52202/068431-2176

[44] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. In *ICLR*.

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

[46] André Hottung, Federico Berto, Chuanbo Hua, Nayeli Gast Zepeda, Daniel Wetzel, Michael Römer, Haoran Ye, Davide Zago, Michael Poli, Stefano Massaroli, Jinkyoo Park, and Kevin Tierney. 2025. VRPAgent: LLM-Driven Discovery of Heuristic Operators for Vehicle Routing Problems. arXiv preprint arXiv:2510.07073.

[47] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *LION (Lecture Notes in Computer Science, Vol. 6683)*. 507–523. doi:10.1007/978-3-642-25566-3_40

[48] Zangir Iklassov, Yali Du, Farkhad Akimov, and Martin Takáč. 2024. Self-Guiding Exploration for Combinatorial Problems. In *NeurIPS*, Vol. 37. 130569–130601. doi:10.52202/079017-4150

[49] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. arXiv preprint arXiv:2001.08361.

[50] Maryam Karimi-Mamaghan, Mehrdad Mohammadi, Patrick Meyer, Amir Mohammad Karimi-Mamaghan, and El-Ghazali Talbi. 2022. Machine learning at the service of meta-heuristics for solving combinatorial optimization problems: A state-of-the-art. *European Journal of Operational Research* 296, 2 (2022), 393–422. doi:10.1016/j.ejor.2021.04.032

[51] Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann. 2019. Automated Algorithm Selection: Survey and Perspectives. *Evolutionary Computation* 27, 1 (2019), 3–45. doi:10.1162/evco_a_00242

[52] Wouter Kool, Herke van Hoof, and Max Welling. 2019. Attention, Learn to Solve Routing Problems!. In *ICLR*. 25 pages.

[53] Lars Kotthoff. 2014. Algorithm Selection for Combinatorial Search Problems: A Survey. *AI Magazine* 35, 3 (2014), 48–60.

[54] Robert Tjarko Lange, Yingtao Tian, and Yujin Tang. 2024. Large Language Models As Evolution Strategies. In *GECCO Companion*. 579–582. doi:10.1145/3638530.3654238

[55] Pedro Larrañaga and Jose A. Lozano (Eds.). 2002. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Kluwer Academic Publishers, Boston, MA. doi:10.1007/978-1-4615-1539-5

[56] Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O. Stanley. 2023. Evolution through Large Models. In *Handbook of Evolutionary Machine Learning*. Springer. doi:10.1007/978-981-99-3814-8_11 arXiv:2206.08896.

[57] Yujian Betterest Li and Kai Wu. 2023. SPELL: Semantic Prompt Evolution based on a LLM. arXiv preprint arXiv:2310.01260.

[58] Fei Liu, Yilu Liu, Qingfu Zhang, Xialiang Tong, and Mingxuan Yuan. 2026. EoH-S: Evolution of Heuristic Set using LLMs for Automated Heuristic Design. In *AAAI*, Vol. 40. 37090–37098. doi:10.1609/aaai.v40i43.41038

[59] Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. 2024. Evolution of Heuristics: Towards Efficient Automatic Algorithm Design Using Large Language Model. In *ICML (PMLR, Vol. 235)*.

[60] Fei Liu, Xialiang Tong, Mingxuan Yuan, and Qingfu Zhang. 2023. Algorithm Evolution Using Large Language Model. arXiv preprint arXiv:2311.15249.

[61] Fei Liu, Yiming Yao, Ping Guo, Zhiyuan Yang, Xi Lin, Zhe Zhao, Xialiang Tong, Kun Mao, Zhichao Lu, Zhenkun Wang, Mingxuan Yuan, and Qingfu Zhang. 2026. A Systematic Survey on Large Language Models for Algorithm Design. *Comput. Surveys* 58, 8 (2026), 1–32. doi:10.1145/3787585

[62] Fei Liu, Rui Zhang, Zhuoliang Xie, Rui Sun, Kai Li, Qinglong Hu, Ping Guo, Xi Lin, Xialiang Tong, Mingxuan Yuan, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. 2024. LLM4AD: A Platform for Algorithm Design with Large Language Model. arXiv preprint arXiv:2412.17287.

[63] Fei Liu, Rui Zhang, Shunyu Yao, Qinglong Hu, Kefeng Zheng, Zhichao Lu, and Qingfu Zhang. 2026. Hierarchical Representations for Cross-task Automated Heuristic Design using LLMs. Preprint, under review (ICLR 2026 submission).

[64] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173. doi:10.1162/tacl_a_00638

[65] Shengcai Liu, Caishun Chen, Xinghua Qu, Ke Tang, and Yew-Soon Ong. 2024. Large Language Models as Evolutionary Optimizers. In *CEC*. doi:10.1109/CEC60901.2024.10611913

[66] Yihong Liu, Junyi Li, Hongyu Lu, Wayne Xin Zhao, and Ji-Rong Wen. 2026. Experience-Driven Reflective Co-Evolution of Prompts and Heuristics for Autonomous Algorithm Design. In *Findings of ACL*. doi:10.18653/v1/2026.findings-acl.245 arXiv:2509.24509.

[67] Manuel López-Ibáñez, Jürgen Branke, and Luís Paquete. 2021. Reproducibility in Evolutionary Computation. *ACM Transactions on Evolutionary Learning and Optimization* 1, 4 (2021), 1–21. doi:10.1145/3466624

[68] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. 2016. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives* 3 (2016), 43–58. doi:10.1016/j.orp.2016.09.002

[69] Chris Lu, Samuel Holt, Claudio Fanconi, Alex J. Chan, Jakob Foerster, Mihaela van der Schaar, and Robert Tjarko Lange. 2024. Discovering Preference Optimization Algorithms with and for Large Language Models. In *NeurIPS*, Vol. 37. 86528–86573. doi:10.52202/079017-2748

[70] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. 2022. Fantastically Ordered Prompts and Where to Find Them: Overcoming Few-Shot Prompt Order Sensitivity. In *ACL*. 8086–8098. doi:10.18653/v1/2022.acl-long.556

[71] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024. Eureka: Human-Level Reward Design via Coding Large Language Models. In *ICLR*.

[72] Andreas Madsen, Siva Reddy, and Sarath Chandar. 2022. Post-hoc Interpretability for Neural NLP: A Survey. *Comput. Surveys* 55, 8, Article 155 (2022), 42 pages. doi:10.1145/3546577

[73] Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. 2022. Locating and Editing Factual Associations in GPT. In *NeurIPS*, Vol. 35.

[74] Elliot Meyerson, Mark J. Nelson, Herbie Bradley, Adam Gaier, Arash Moradi, Amy K. Hoover, and Joel Lehman. 2024. Language Model Crossover: Variation through Few-Shot Prompting. *ACM Transactions on Evolutionary Learning and Optimization* (2024). doi:10.1145/3694791

[75] Moran Mizrahi, Guy Kaplan, Dan Malkin, Rotem Dror, Dafna Shahaf, and Gabriel Stanovsky. 2024. State of What Art? A Call for Multi-Prompt LLM Evaluation. *Transactions of the Association for Computational Linguistics* 12 (2024), 933–949. doi:10.1162/tacl_a_00681

[76] Alberto Moraglio, Krzysztof Krawiec, and Colin G. Johnson. 2012. Geometric Semantic Genetic Programming. In *PPSN (LNCS, Vol. 7491)*. 21–31. doi:10.1007/978-3-642-32937-1_3

[77] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat

Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. arXiv preprint arXiv:2506.13131.

[78] Pham Vu Tuan Dat, Long Doan, and Huynh Thi Thanh Binh. 2025. HSEvo: Elevating Automatic Heuristic Design with Diversity-Driven Harmony Search and Genetic Algorithm Using LLMs. In *AAAI*, Vol. 39. 26931–26938. doi:10.1609/aaai.v39i25.34898

[79] Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Larivière, Alina Beygelzimer, Florence d'Alché Buc, Emily Fox, and Hugo Larochelle. 2021. Improving Reproducibility in Machine Learning Research (A Report from the NeurIPS 2019 Reproducibility Program). *Journal of Machine Learning Research* 22, 164 (2021), 1–20.

[80] Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. 2023. Automatic Prompt Optimization with "Gradient Descent" and Beam Search. In *EMNLP*. doi:10.18653/v1/2023.emnlp-main.494

[81] John R. Rice. 1976. The Algorithm Selection Problem. In *Advances in Computers*. Vol. 15. Academic Press, 65–118.

[82] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. 2024. Mathematical discoveries from program search with large language models. *Nature* 625 (2024), 468–475. doi:10.1038/s41586-023-06924-6

[83] Stefan Ropke and David Pisinger. 2006. An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows. *Transportation Science* 40, 4 (2006), 455–472. doi:10.1287/trsc.1050.0135

[84] Günter Rudolph. 1994. Convergence Analysis of Canonical Genetic Algorithms. *IEEE Transactions on Neural Networks* 5, 1 (1994), 96–101. doi:10.1109/72.265964

[85] Maxim Sakharov. 2026. A Framework for Integrating Large Language Models into Memetic Algorithms. *Biomimetics* 11, 6 (2026), 383. doi:10.3390/biomimetics11060383

[86] Roberto Santana. 2017. Gray-box optimization and factorized distribution algorithms: where two worlds collide. arXiv preprint arXiv:1707.03093.

[87] Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. 2024. Quantifying Language Models' Sensitivity to Spurious Features in Prompt Design or: How I Learned to Start Worrying About Prompt Formatting. In *ICLR*.

[88] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. 2016. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proc. IEEE* 104, 1 (2016), 148–175. doi:10.1109/JPROC.2015.2494218

[89] Lin Shi, Chiyu Ma, Wenhua Liang, Xingjian Diao, Weicheng Ma, and Soroush Vosoughi. 2025. Judging the Judges: A Systematic Study of Position Bias in LLM-as-a-Judge. In *IJCNLP-AACL*. 292–314. doi:10.18653/v1/2025.ijcnlp-long.18

[90] Christopher Strachey. 2000. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation* 13, 1–2 (2000), 11–49. doi:10.1023/A:1010000313106

[91] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and Policy Considerations for Deep Learning in NLP. In *ACL*. 3645–3650. doi:10.18653/v1/P19-1355

[92] Yiwen Sun, Furong Ye, Xianyin Zhang, Shiyu Huang, Bingzhen Zhang, Ke Wei, and Shaowei Cai. 2026. AutoSAT: Automatically Optimize SAT Solvers via Large Language Models. *Journal of Artificial Intelligence Research* 86 (2026). doi:10.1613/jair.1.20499 arXiv:2402.10705.

[93] Anja Šurina, Amin Mansouri, Lars C. P. M. Quaedvlieg, Amal Seddas, Maryna Viazovska, Emmanuel Abbe, and Caglar Gulcehre. 2025. Algorithm Discovery With LLMs: Evolutionary Search Meets Reinforcement Learning. In *COLM*.

[94] El-Ghazali Talbi. 2009. *Metaheuristics: From Design to Implementation*. Wiley. doi:10.1002/9780470496916

[95] El-Ghazali Talbi. 2021. Machine Learning into Metaheuristics: A Survey and Taxonomy. *Comput. Surveys* 54, 6, Article 129 (2021), 32 pages. doi:10.1145/3459664

[96] Dirk Thierens. 2005. An adaptive pursuit strategy for allocating operator probabilities. In *GECCO*. 1539–1546. doi:10.1145/1068009.1068251

[97] Niki van Stein and Thomas Bäck. 2025. LLaMEA: A Large Language Model Evolutionary Algorithm for Automatically Generating Metaheuristics. *IEEE Transactions on Evolutionary Computation* 29, 2 (2025), 331–345. doi:10.1109/TEVC.2024.3497793

[98] Leonardo Vanneschi, Mauro Castelli, and Sara Silva. 2014. A survey of semantic methods in genetic programming. *Genetic Programming and Evolvable Machines* 15, 2 (2014), 195–214. doi:10.1007/s10710-013-9210-0

[99] Haorui Wang, Marta Skreta, Cher-Tian Ser, Wenhao Gao, Lingkai Kong, Felix Strieth-Kalthoff, Chenru Duan, Yuchen Zhuang, Yue Yu, Yanqiao Zhu, Yuanqi Du, Alán Aspuru-Guzik, Kirill Neklyudov, and Chao Zhang. 2025. Efficient Evolutionary Search Over Chemical Space with Large Language Models. In *ICLR*.

[100] Yidan Wang, Jiayin Wang, and Zhiwei Chu. 2025. Multi-agent large language models as evolutionary optimizers for scheduling optimization. *Computers & Industrial Engineering* 206 (2025), 111197. doi:10.1016/j.cie.2025.111197

[101] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *NeurIPS*, Vol. 35. 24824–24837.

[102] Brandon T. Willard and Rémi Louf. 2023. Efficient Guided Generation for Large Language Models. arXiv preprint arXiv:2307.09702.
[103] Glynn Winskel. 1993. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, Cambridge, MA.
[104] David H. Wolpert and William G. Macready. 1997. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation* 1, 1 (1997), 67–82. doi:10.1109/4235.585893
[105] Xingyu Wu, Sheng-Hao Wu, Jibin Wu, Liang Feng, and Kay Chen Tan. 2025. Evolutionary Computation in the Era of Large Language Model: Survey and Roadmap. *IEEE Transactions on Evolutionary Computation* 29, 2 (2025), 534–554. doi:10.1109/TEVC.2024.3506731
[106] Sang Michael Xie, Aditi Raghunathan, Percy Liang, and Tengyu Ma. 2022. An Explanation of In-context Learning as Implicit Bayesian Inference. In *ICLR*.
[107] Cheng Xu, Shuhao Guan, Derek Greene, and M-Tahar Kechadi. 2024. Benchmark Data Contamination of Large Language Models: A Survey. arXiv preprint arXiv:2406.04244.
[108] Hanwei Xu, Yujun Chen, Yulun Du, Nan Shao, Yanggang Wang, Haiyu Li, and Zhilin Yang. 2022. GPS: Genetic Prompt Search for Efficient Few-shot Learning. In *EMNLP*. doi:10.18653/v1/2022.emnlp-main.559
[109] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. 2024. Large Language Models as Optimizers. In *ICLR*.
[110] Xianliang Yang, Ling Zhang, Haolong Qian, Lei Song, and Jiang Bian. 2025. HeurAgenix: Leveraging LLMs for Solving Complex Combinatorial Optimization Challenges. arXiv preprint arXiv:2506.15196.
[111] Shunyu Yao, Fei Liu, Xi Lin, Zhichao Lu, Zhenkun Wang, and Qingfu Zhang. 2025. Multi-Objective Evolution of Heuristic Using Large Language Model. In *AAAI*, Vol. 39. 27144–27152. doi:10.1609/aaai.v39i25.34922
[112] Yiming Yao, Fei Liu, Ji Cheng, and Qingfu Zhang. 2024. Evolve Cost-aware Acquisition Functions Using Large Language Models. In *PPSN (LNCS, Vol. 15149)*. doi:10.1007/978-3-031-70068-2_23
[113] Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. 2024. ReEvo: Large Language Models as Hyper-Heuristics with Reflective Evolution. In *NeurIPS*, Vol. 37. 43571–43608. doi:10.52202/079017-1381
[114] Huigen Ye, Hua Xu, An Yan, and Yaoyang Cheng. 2025. Large Language Model-driven Large Neighborhood Search for Large-Scale MILP Problems. In *ICML (Proceedings of Machine Learning Research, Vol. 267)*. 72131–72180.
[115] Qinyuan Ye, Maxamed Axmed, Reid Pryzant, and Fereshte Khani. 2024. Prompt Engineering a Prompt Engineer. In *Findings of ACL*. doi:10.18653/v1/2024.findings-acl.21
[116] Zhuoli Yin, Yi Ding, Reem Khir, and Hua Cai. 2025. ViTSP: A Vision Language Models Guided Framework for Large-Scale Traveling Salesman Problems. arXiv preprint arXiv:2509.23465.
[117] Cunxi Yu, Rongjian Liang, Chia-Tung Ho, and Haoxing Ren. 2025. Autonomous Code Evolution Meets NP-Completeness. arXiv preprint arXiv:2509.07367.
[118] Shaofeng Zhang, Shengcai Liu, Ning Lu, Jiahao Wu, Ji Liu, Yew-Soon Ong, and Ke Tang. 2025. LLM-Driven Instance-Specific Heuristic Generation and Selection. arXiv preprint arXiv:2506.00490.
[119] Yisong Zhang, Ran Cheng, Guoxing Yi, and Kay Chen Tan. 2025. A Systematic Survey on Large Language Models for Evolutionary Optimization: From Modeling to Solving. arXiv preprint arXiv:2509.08269.
[120] Jie Zhao and Kang Hao Cheong. 2025. Visual Evolutionary Optimization on Graph-Structured Combinatorial Problems With MLLMs: A Case Study of Influence Maximization. *IEEE Transactions on Evolutionary Computation* (2025). doi:10.1109/TEVC.2025.3598266 Early access.
[121] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate Before Use: Improving Few-shot Performance of Language Models. In *ICML (PMLR, Vol. 139)*. 12697–12706.
[122] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *NeurIPS*, Vol. 36. 46595–46623. doi:10.52202/075280-2020
[123] Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. 2025. Monte Carlo Tree Search for Comprehensive Exploration in LLM-Based Automatic Heuristic Design. In *ICML (PMLR, Vol. 267)*. 78338–78373.
[124] Rui Zhong, Abdelazim G. Hussien, Jun Yu, and Masaharu Munetomo. 2025. LLMOA: A novel large language model assisted hyper-heuristic optimization algorithm. *Advanced Engineering Informatics* 64 (2025), 103042. doi:10.1016/j.aei.2024.103042
[125] Xun Zhou, Xingyu Wu, Liang Feng, Zhichao Lu, and Kay Chen Tan. 2025. Design Principle Transfer in Neural Architecture Search via Large Language Models. In *AAAI*, Vol. 39. 23000–23008. doi:10.1609/aaai.v39i21.34463
[126] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2023. Large Language Models Are Human-Level Prompt Engineers. In *ICLR*.

A Reusable templates

This appendix provides a copyable scaffold that instantiates the anatomy of Section 4 (Fig. 5) and the build loop of Algorithm 1: first the prompt the operator sends, then the code that wraps it.

The prompt template. The template consists of the four slots mentioned in Section 4; the angle-bracketed fields are the only parts that change per problem. **[context]** (lines 1–5) fixes the task and the legal building blocks — line 4 declares what the operator emits (a solution, an operator, or a program), and line 5 bounds the output to a vocabulary the validator can check. **[conditioning]** (lines 7–12) is the load-bearing channel (Section 3): the representation R and constraints C (lines 8–9) stay fixed, whereas the incumbent, scores, and diagnostics (lines 10–11) are the dynamic, search-state-derived content that makes the operator adaptive rather than one-shot. **[instruction]** (lines 14–16) states the single move and the two invariants the validator enforces (preserve R; do not violate C). **[format]** (lines 18–26) fixes the `CANDIDATE` envelope (lines 20–26): Keeping it stable across problems is exactly what lets the parser remain unchanged, so only R and the feasibility test are problem-specific.

```
1  [context]
2  Problem type: <name>
3  Objective direction: <minimize|maximize>
4  Candidate artifact: <solution|operator|program>
5  Available primitives: <allowed moves/functions/libraries>
6
7  [conditioning]
8  Representation R: <exact syntax>
9  Constraints C: <hard feasibility rules>
10 Incumbent or parents: <current artifact(s)>
11 Scores and diagnostics: <evaluator output and recent failures>
12 Structural notes: <optional decomposition or domain facts>
13
14 [instruction]
15 Emit exactly one new candidate. Preserve R. Use only the available
16 primitives. If diagnostics are present, target them without violating C.
17
18 [format]
19 Return only this envelope:
20 CANDIDATE
21 id: <short_id>
22 artifact_type: <solution|operator|program>
23 representation: <name of R>
24 payload:
25 <candidate>
26 END_CANDIDATE
```

The reference loop. The function below realizes Algorithm 1; its arguments (lines 1–2) separate what a new problem must supply from the fixed control flow. `llm` samples a completion; `evaluator` returns the objective score f; `accept` is the selection rule (e.g. "keep iff the score improves"); `incumbent` is the starting artifact; `budget` bounds outer search steps; and `retries` caps bounded-repair samples within a step. The problem binding is `spec` — four methods that adapt the template above: `render` fills the slots based on the incumbent, its cached score, and the feedback log; `parse` reads the envelope and the payload in representation R; `feasible` checks the constraints C and raises a `ValueError` with a diagnostic on failure; and `repair_prompt` rebuilds the full prompt with the failing output and diagnostic appended for the next attempt. The inner loop is the bounded repair of Algorithm 1: ill-formed output is re-prompted up to `retries` times and then skipped; a valid candidate is scored and either accepted or rejected, and the outcome is appended to `feedback` so the next prompt is conditioned on it.

```
1  def build_and_validate(llm, evaluator, accept, incumbent, spec,
```

```
                    budget, retries):
    """Reference loop; spec supplies render(), parse(), feasible(), repair_prompt()."""
    cur = best = incumbent
    score_cur = score_best = evaluator(cur)
    feedback = []
    for _ in range(budget):
        prompt = spec.render(cur, score_cur, feedback)
        ok, cand, err = False, None, None
        for _ in range(retries + 1):
            text = llm.sample(prompt)
            try:
                cand = spec.parse(text)          # schema + syntax
                spec.feasible(cand)              # domain feasibility
                ok = True
                break
            except ValueError as exc:
                err = str(exc)
                prompt = spec.repair_prompt(prompt, text, err)
        if not ok:
            feedback.append(("invalid", err))
            continue
        score_c = evaluator(cand)
        gap = score_c - score_cur
        if accept(cand, score_c, cur, score_cur):
            cur, score_cur = cand, score_c
            if score_cur < score_best:        # best ever (min.)
                best, score_best = cur, score_cur
            feedback.append(("accepted", score_c, gap))
        else:
            feedback.append(("rejected", score_c, gap))
    return best
```

A concrete binding. For the TSP operator of Section 4, `spec.parse` extracts the permutation from the `payload`; `spec.feasible` checks that every city appears exactly once (raising "city k duplicated, city j missing" otherwise); `evaluator` returns the closed-tour length; and `accept` keeps a candidate only when that length decreases. Re-binding `parse`/`feasible`/`evaluator` retargets the identical loop to bin packing, job-shop scheduling, or program search without touching the control flow: the operator changes only in what it is conditioned on and what counts as feasible.

For more examples, visit the repository: https://camilochs.github.io/semantic-turn-metaheuristics.

B Placing every method: a drop-channel audit

Table 3 applies the classification rule of Section 3 to each method in Table 2: it lists the channels present in the variation prompt and names the *dominant* one—the channel intended to carry the operative search signal, audited where possible by removing or neutralizing that channel while preserving a valid scaffold—with a one-line placement justification. This makes the placements reproducible and resolves the cases where a method *emits* code yet has a `Linguistic` dominant channel (*EoH*, *ReEvo*, *MEoH*): the code channel is shared with the rest of the family, so the dominant channel is the natural-language idea or reflection that steers it.

Validation of the placement rule. Two checks accompany the companion code. *(i) Reproducibility.* Three independent LLM coders — one instance each from three model families (Claude Fable 5, GPT-5.5, DeepSeek-V4-Pro), queried separately and shown only the placement rule and the channel descriptions of Table 3 (method names visible; coders instructed to reason from the rule alone) — classified the twelve methods of Table 2. The three coders coincide on nine of the twelve; chance-corrected agreement is Gwet's AC1 = 0.73. The divergences fall on the placements whose channel description is least determinate: OPRO and LMX, whose bare "solution" descriptions underdetermine the representation, and ReEvo, where the steering-reflection and code-substrate readings compete. Of these, OPRO carries a boundary annotation in Table 3. For EvoPrompt, all three coders

Table 3. Drop-channel classification audit for the methods of Table 2. The table column **Dominance** is the source-described operative channel, audited where possible by removing or neutralizing that signal (`Numeric`, `Symbolic`, `Linguistic`); the other channels present are listed but not ranked.

Method	Channels present	Dominance (ablation/-source)	Drop-channel reasoning
OPRO	(solution, score) trajectory	`Numeric` (boundary)	drop the score trajectory and no signal steers the next proposal
EvoPrompt	parent prompt text, scores	`Numeric` (boundary)	scored parents drive selection, while parent text is the artifact varied; boundary label retained
LMX	parent solutions (few-shot)	`Numeric`	drop the parents and there is no crossover to perform
FunSearch	sampled high-scoring programs, scores	`Symbolic`	drop the program text and there is nothing to mutate
LLaMEA	algorithm code, score, error note	`Symbolic`	drop the code and no algorithm remains; the error note is annotated
MCTS-AHD	program, tree-search state, scores	`Symbolic`	drop the code and the tree explores nothing executable
AlphaEvolve	codebase, evaluator scores	`Symbolic`	drop the code and there is no artifact to evolve
EoH	NL idea, code, scores	`Linguistic`	neutralize the idea and it collapses toward FunSearch-style code mutation
ReEvo	code, reflection, scores	`Linguistic`	neutralize the reflection and it collapses toward a code-only AHD loop
HSEvo	NL idea, code population, scores, diversity pressure	`Linguistic`	neutralize the idea and diversity-seeking rationale, and the loop loses its language-steered population-diversity mechanism
MEoH	NL idea, code, multi-obj. scores	`Linguistic`	neutralize the idea and the multi-objective search loses its steering rationale
LAPT	NL design principles	`Linguistic`	drop the principles and no transferable signal remains

read the varied prompt text as linguistic content, whereas Table 3 reports the scored trajectory as the load-bearing channel (Numeric, boundary): the placement of solution-as-text methods depends on whether the varied text is read as an encoding or as operative language, which is why the row carries a boundary annotation. The labeling is therefore largely reproducible, and it is contested exactly where the channel description leaves the representation open. *(ii) Load-bearing test.* A drop-channel ablation on an EoH-style operator for online bin packing (companion `ablation.py`) compared candidate heuristics generated *with* versus *without* the natural-language idea, holding the parent code and scores fixed. On this easy task both conditions reached the optimum (mean gap 0%; 6/6 candidates optimal), so the ablation did not demonstrate performance necessity for the natural-language channel on that task: the operator recovers the known best-fit heuristic with or without it. We nevertheless report EoH as Linguistic because the idea is the source-described design-intent signal that structurally distinguishes it from code-only automatic heuristic design. Together the checks say placement is reproducible, but *how much* a present channel contributes is task-dependent — consistent with the no-law stance of Section 3: we report the channel a method *uses* to steer variation, not a universal claim about the channel that *determines* performance.

C Extended coverage map

Table 4 places a broader set of LLM-operator methods on the lens, beyond the representative methods of the body, to confirm that the selection of Section 5 is representative of the field's breadth. Each entry is read against its primary source. Entries marked *adjacent* fall outside the lens — they translate a specification rather than *vary* a candidate — and are listed only as foils. This map illustrates breadth, with the systematic census provided by [25]; we report each method's *dominant* channel, persistence descriptor, and channels it draws on (Section 3).

Table 4. Extended coverage: LLM variation operators by dominant conditioning channel and persistence. Entries are read against their primary sources. Channel and persistence assignments follow Section 3; this is not a systematic census.

Method	Dominant Conditioning	Persistence	Conditions on → emits
EvoLLM [54]	`Numeric`	`Transient`	fitness-ranked solution vectors → distribution/candidate update
LMEA [65]	`Numeric`	`Transient`	parent tours + tour lengths (TSP instantiation) → offspring tour (LLM crossover/mutation)
DiscoPOP [69]	`Symbolic`	`Amortized`	prior objective/loss code + eval metrics → preference-optimization loss; held-out-task reuse
ELM [56]	`Symbolic`	`Transient`	parent code + edit instruction (diff model) → program; adjacent background anchor (Section 5)
EvoPrompting [20]	`Symbolic`	`Amortized`	parent architecture code + fitness → NN architecture code
GI-LLM [12]	`Symbolic`	`Transient`	source code + edit type → code patch (GI mutation)
AutoSAT [92]	`Symbolic`	`Amortized`	parent CDCL heuristic code + hints → SAT-solver heuristic
EvoTune [93]	`Symbolic`	`Amortized`	parent programs + RL reward into weights → heuristic program
CoEvo [36]	`Symbolic`	`Amortized`	prior symbolic solutions + knowledge library → math/code solution
VRPAgent [46]	`Symbolic`	`Amortized`	operator code + LNS feedback → destroy/repair operators
LLM-LNS [114]	`Symbolic`	`Amortized`	MILP state + neighborhood rule + fitness → neighborhood-selection heuristic
InstSpecHH [118]	`Symbolic`	`Amortized`	instance features + fitness → per-subclass heuristic + selection
SATLUTION [117]	`Symbolic`	`Amortized`	solver repo + correctness/runtime feedback (agentic) → solver code
Eureka [71]	`Linguistic` (boundary)	`Amortized`	env. code + reward reflection + parent reward code → reward-function code; reflection steers code edits
QDAIF [10]	`Linguistic`	`Amortized`	few-shot NL parents + LLM feedback → NL text
EvolCAF [112]	`Linguistic`	`Transfer`	NL design idea + code (EoH) → cost-aware acquisition function
L-AutoDA [35]	`Linguistic`	`Amortized`	NL idea + code (EoH) → adversarial-attack algorithm
SGE [48]	`Linguistic`	`Transient`	NL problem description + thought trajectories → solution via heuristics
EvoPH [66]	`Linguistic`	`Transfer`	co-evolved NL prompts + heuristic code → heuristics + prompts
MTHS [63]	`Linguistic`	`Transfer`	task-agnostic + task-specific programs, cross-task transfer → metaheuristic + heuristic
MAEF [100]	`Linguistic`	`Transient`	role-specialized agents + eval feedback → schedules
APE [126]	`Linguistic`	`Amortized`	task demonstrations → prompt (instruction induction)
ProTeGi/APO [80]	`Linguistic`	`Amortized`	prompt + error minibatch (NL "gradient") → prompt
Promptbreeder [29]	`Linguistic`	`Amortized`	prompt population + self-evolved mutation-prompts → prompts
PE2 [115]	`Linguistic`	`Amortized`	prompt + examples + meta-prompt → prompt
GPS [108]	`Linguistic`	`Amortized`	seed prompts + scores (T5 variation) → prompt
SPELL [57]	`Linguistic`	`Amortized`	prompts + fitness → prompt (whole-text generation)
ViTSP [116]	multimodal	`Amortized`	rendered image of the instance (VLM) → subproblem selection in an LNS loop
VEO [120]	multimodal	`Transient`	rendered image of the network solution (MLLM) → in-loop crossover/mutation operators for influence maximization
OptiMUS [4]	—	—	NL problem description → MILP model + solver code (*translates*, does not vary)