Review

Not peer-reviewed version

Automated Design of Agentic Systems: A Survey of Algorithms for Searching, Optimizing, and Evolving LLM Agents, Workflows, and Prompts

Maksim Madžar * and Igor Mekterović

Posted Date: 2 June 2026

doi: 10.20944/preprints202606.0238.v1

Keywords: automated design of agentic systems; LLM agents; agentic workflows; prompt optimization; neural architecture search; AutoML; self-improving agents

 doi:10.20944/preprints202606.0238.v1

Review

Automated Design of Agentic Systems: A Survey of Algorithms for Searching, Optimizing, and Evolving LLM Agents, Workflows, and Prompts

Maksim Madžar * and Igor Mekterović

Faculty of Electrical Engineering and Computing, University of Zagreb
* Correspondence: madzarmaksim@gmail.com

Abstract

Building agentic systems around frozen foundation models remains largely a craft: chains of thought, reflection loops, tool calls, and multi-agent topologies are assembled by hand. Automated Design of Agentic Systems (ADAS) recasts that craft as a search problem. In this paper we survey the ADAS literature from 2022 to 2026 through a unifying four-axis framework that re-casts the three classical pillars of neural architecture search (NAS) in an agentic setting: the optimization target (the prompt, the parameters of a compound system, the workflow topology, a modular cell, or full code), the search strategy (LLM-as-optimizer, textual gradients, evolutionary and quality-diversity search, MCTS, Bayesian or surrogate methods, and RL/DPO), the representation (a natural-language string, a modular DSL, a graph, or code), and the feedback signal (a scalar, a preference, a natural-language critique, a surrogate, or novelty). We classify thirty-three methods along these axes, spanning 2022 to 2026 and including the recent surge of 2026 work, trace two structural tensions—expressiveness versus searchability and feedback richness versus credit assignment—and examine evaluation practice, including cost, transfer, contamination, and reward hacking. We conclude with the safety problems that recursive self-improvement makes concrete and the open directions they impose.

Keywords: automated design of agentic systems; LLM agents; agentic workflows; prompt optimization; neural architecture search; AutoML; self-improving agents

---

1. Introduction

Modern agents are compound systems: a foundation model is embedded as a module within a wider orchestration [18]. The components of that orchestration—prompt templates, reasoning patterns, tool definitions, and inter-agent topology—are almost all hand-designed, even though the history of machine learning suggests that hand-built solutions are eventually displaced by learned ones, with search and computation overriding embedded human knowledge [44]. Automated Design of Agentic Systems (ADAS) makes this explicit, treating the design of agentic systems as a search problem: an algorithm finds systems within a search space so as to optimize an evaluation function [18]. This approach traces its roots to neural architecture search (NAS) and AutoML, which reformulated network design around three pillars—the search space, the search strategy, and performance estimation [10]—and to Clune's vision of AI-generating algorithms, in which meta-learning learns the very mechanism that produces intelligence [6]. We organize the field with a four-axis framework that adapts the NAS triad to agents: the optimization target, the search strategy, the representation, and the feedback signal. This separates questions that named systems conflate, placing prompt optimizers (which build on chain-of-thought reasoning patterns [50]), compound-system compilers, workflow and topology searchers, modular architecture search, and recursively self-improving agents into a single comparative space. Our scope is the optimization of scaffolds around largely frozen models at design time; we touch weight training only where it borders on scaffold search. We review methods from 2022 to 2026, their evaluation, and the safety questions that self-improvement raises.

 doi:10.20944/preprints202606.0238.v1

2. Foundations and Problem Definition

> *"The history of machine learning teaches us that hand-designed solutions are eventually replaced by learned ones." [18]*

Building agents rests on a compositional premise: a foundation model (FM) is embedded as a *module* within a wider orchestration that plans, calls tools, reflects, and iterates. Hu, Lu, and Clune name the problem of automating such design ADAS [18], extrapolating from features and architectures to agents themselves. Because the community has no consensus on terminology, they define agents as "agentic systems that involve foundation models (FMs) as modules within a workflow to solve tasks by planning, using tools, and carrying out multiple, iterative steps of processing" [18]. We read this in layers: an **LLM agent** places an LLM in a control loop with memory and, optionally, tools; an **agentic workflow** is a graph/program of FM calls, control flow, and intermediate state; an **agentic system** is the complete object—the workflow together with prompts, tool definitions, and configuration. ADAS targets the automatic design of precisely that complete object.

The landscape before ADAS is a catalog of hand-designed building blocks: Chain-of-Thought [50] and Self-Consistency [48] for reasoning; Reflexion [43] for self-critique; ReAct [54] and Toolformer [40] for tool use; multi-agent debate [9]; and role-based pipelines such as MetaGPT [16], ChatDev [36], and the programmable AutoGen [51]. The advance of ADAS is to treat that space as something to be *searched*.

2.1. The Formal Problem and the Bet on Code Space

In direct analogy with AutoML and neural architecture search (NAS), "ADAS uses a **search algorithm** to discover agentic systems within a **search space** that optimize an **evaluation function**" [18]. The **search space** "defines which agentic systems can be represented"; prompt-only methods that "change only the textual prompts" cannot represent agents with different workflows [18], with graph representations and fixed-network representations arising as intermediate cases. The **search algorithm** explores this often unbounded space, so it "must consider the exploration–exploitation trade-off" [18]. The **evaluation function** scores candidates by performance, cost, latency, or safety, most often by validation-set accuracy [18]. This decomposition sorts later methods by target, search, representation, and feedback signal.

The choice of representation is crucial. Because Python is Turing-complete, "searching within code space theoretically lets the ADAS algorithm discover *any* possible agentic system" [18], thereby subsuming both prompt-only and graph-only spaces. Code is human-readable, lets search reuse existing human frameworks, and exploits the FMs' programming skill [18]. The trade-off is that bespoke spaces carry sample-efficient priors that code sacrifices for expressiveness—and that is the central tension of the entire field.

Meta Agent Search is "one of the first algorithms in ADAS to enable full design in code space" [18]. A candidate agent is a `forward(task)` function that the meta agent programs within a framework of roughly 100 lines, "similar to FunSearch" [18,38]. The operator: FMs "as meta agents iteratively program interestingly new agents based on an ever-growing archive of previous discoveries" [18]—the archive is initialized with CoT/Self-Refine, an idea is generated, then archive-conditioned code, two self-reflections on novelty are run, it is evaluated (with up to five rounds of debugging on errors), and added to the archive. The feedback signal combines scalar task scores with an open-ended pressure toward "interestingness" [18].

Empirically it runs on reasoning and coding benchmarks (ARC-AGI, DROP, MGSM, MMLU, GPQA) with GPT-4 as the meta agent and GPT-3.5 executing the agents, "substantially" outperforming CoT, CoT-SC, Self-Refine, LLM-Debate, and Quality-Diversity (+13.6 F1 on DROP, +14.4% on MGSM), with discovered agents transferring across domains and models [18]. ADAS embodies Sutton's "bitter lesson" that "general methods that leverage computation are ultimately the most effective" [44], lifting NAS/AutoML one level higher [10], and realizes pillar (1)—"meta-learning architectures"—of Clune's **AI-GAs** vision [6,18]. Complementing it, **AIOS** isolates LLM services into a kernel responsible for

 doi:10.20944/preprints202606.0238.v1

scheduling, memory, and access control [29]; it formalizes the execution substrate and addresses executing "untrusted, model-generated code" [18,29]. The foundations expose four enduring open problems: the real search space is the FM's prior over code, not all programs; the exploration–exploitation trade-off remains unresolved; evaluation is compute-bounded and prone to overfitting; and executing novel code is a structural safety problem. These are the axes along which later works differ from one another.

3. A Unifying Framework for the Automated Design of Agents

The three pillars of NAS—search space, search strategy, and performance estimation [10]—we recast into four axes that separate what named ADAS systems merge. Tables~1–2 map the surveyed methods onto these axes. The correspondence is exact but refined: NAS's *search space* splits into our *target* and *representation*, its *performance estimation* generalizes to our *feedback signal*, and its *search strategy* is retained. This split is justified because in ADAS the target and the representation vary independently: two methods can share the same code-based representation yet have one target prompts and the other entire systems, or they can share the same target (prompts) with a natural-language representation in one case and a DSL in the other. We fit cross-cutting sub-targets such as memory and tool synthesis into these four axes rather than elevating them into a fifth, since each is searched within one of them (AgentSquare, for instance, searches a memory module within its four-slot cell).

The first axis, the optimization TARGET, asks what changes and forms a ladder of ever-wider scope (Figure 1): a single instruction; the instructions and demonstration examples of a fixed compound program; the who-calls-whom topology of a workflow; a modular cell of interchangeable components; and fully Turing-complete code. Hu et al. argue that a code-level target subsumes all the others, because code space can express any agentic system [18]; the price is searchability.

The second axis, the SEARCH STRATEGY, asks how candidates are proposed. The minimalist case is LLM-as-optimizer, which reads scored solutions and proposes better ones with no explicit update rule [53]. Textual gradients verbalize errors and edit in the opposite semantic direction [5,35,58]. Population methods emphasize exploration and open-endedness, sometimes evolving the operators themselves [11,12,31]. MCTS imposes a principled exploration–exploitation balance [47,63]. Surrogate methods confront the cost of evaluation directly [33,39,42]. RL and DPO methods turn the operator into a learnable policy [49,60,67].

The third axis, the REPRESENTATION, asks how a candidate is encoded: as a natural-language string, a DSL of typed blocks, a graph of nodes and edges, or arbitrary code. The representation bounds reach: prompt-only encodings cannot alter control flow [18], whereas code can, at the price of a practically unbounded space that can be navigated only by leaning on the LLM's prior programming instinct.

The fourth axis, the FEEDBACK SIGNAL, asks what drives selection: a scalar measure, a preference, a critique, a surrogate prediction, or a novelty pressure. Most methods use execution-based scalars; the rich-feedback line replaces them with critiques used as gradients [58], and preference-based methods exploit the magnitude of the reward [49].

Two tensions run through all the axes. Expressiveness versus searchability: Turing-complete targets can express anything but require operator templates, archives, or surrogates, whereas modular DSLs and supernets sacrifice reach for sample efficiency and transferability [18,42,60]. Feedback richness versus credit assignment: a scalar feedback signal is general but assigns credit only end-to-end, whereas richer signals localize blame but rely on hallucinated, unguaranteed critiques, and almost no method performs process-level credit assignment across a multi-step framework [27]. Tables 1 and 2 list methods whose primary contribution is a *search or optimization procedure over agent designs*; hand-designed frameworks (Chain-of-Thought, ReAct, MetaGPT) and weight-training methods appear only as context.

 doi:10.20944/preprints202606.0238.v1

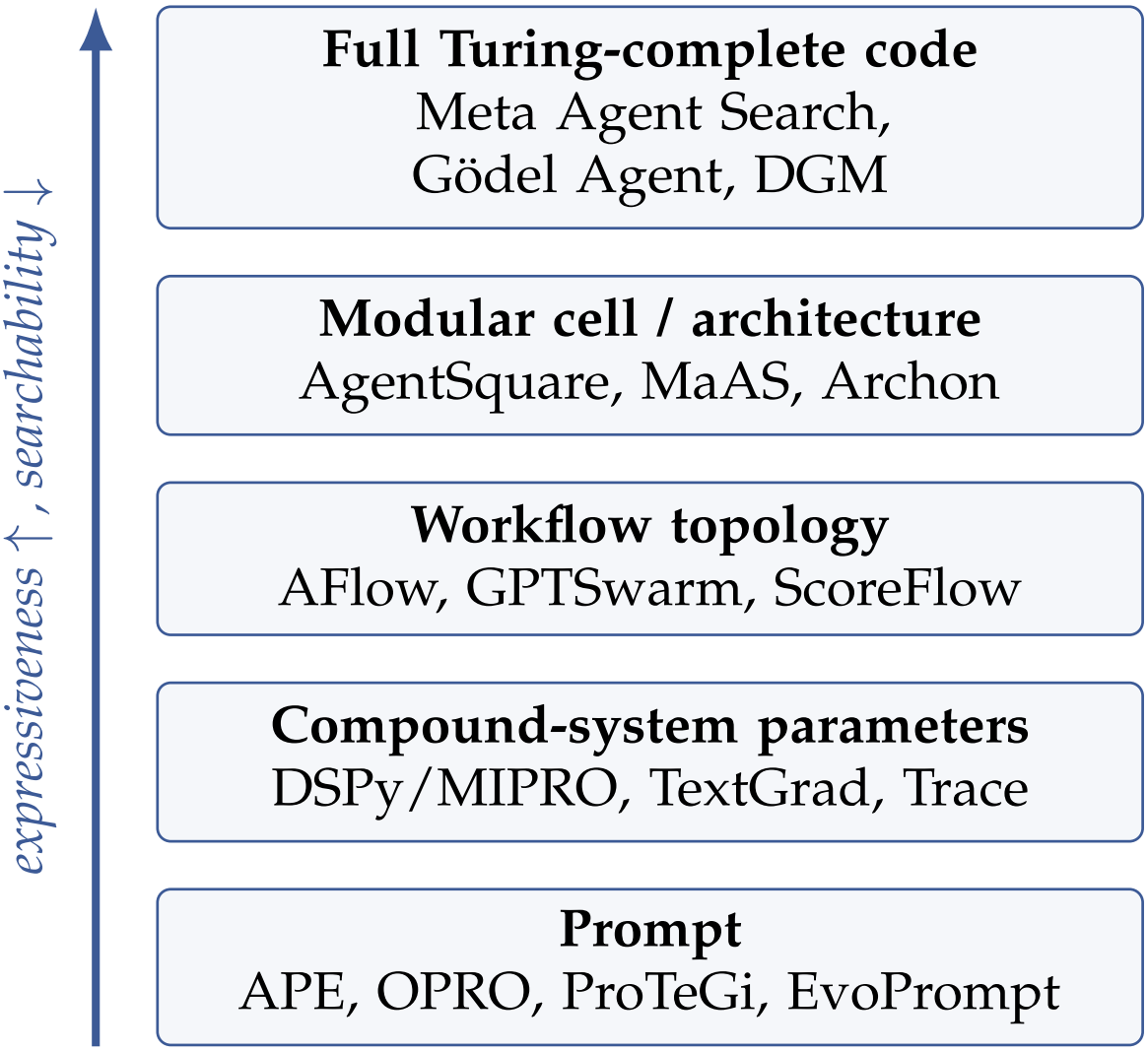

Figure 1. *The ladder of optimization targets.* ADAS methods are organized by *what* they search, from a single prompt all the way to full Turing-complete code. Climbing the ladder gains expressiveness but enlarges the search space and raises evaluation cost—the central expressiveness-versus-searchability tension of this survey. Generator-based workflow methods (e.g. ScoreFlow) occupy the workflow-topology rung even though they produce programs.

 doi:10.20944/preprints202606.0238.v1

Table 1. Taxonomy of representative methods for the automated design of agentic systems, classified by optimization target, search strategy, representation, and feedback signal (part 1 of 2: methods for prompts, compound systems, and workflows). Methods spanning 2022–2026 are listed together, including the 2026 entries. The gains cited in the text rest on heterogeneous baselines and benchmark sets and are not directly comparable across rows. Year denotes the first public/arXiv release.

Method	Year	Target	Search strategy	Representation	Feedback signal	Ref.
APE	2022	Prompt	LLM-as-proposer + score filtering + Monte Carlo resampling	Natural-language string	Scalar score (accuracy / log-likelihood)	[66]
OPRO	2023	Prompt	LLM-as-optimizer over (solution, score) history	Natural-language string	Scalar score on a training subset	[53]
ProTeGi / APO	2023	Prompt	Textual-gradient descent + beam search (UCB bandit)	Natural-language string	Natural-language critique (textual gradient from an error set)	[35]
EvoPrompt	2023	Prompt	Evolutionary (GA + differential evolution)	Natural-language string (genome)	Population fitness (batch score)	[12]
Promptbreeder	2023	Prompt (+ mutation operators)	Self-referential evolutionary (tournament GA)	Natural-language strings (prompt + mutation prompt)	Fitness on a held-out set	[11]
PromptAgent	2023	Prompt	MCTS / planning over an MDP (UCT)	Natural-language string	Natural-language error feedback + simulated reward	[47]
GEPA	2025	Prompt / compound	Reflective evolutionary (Pareto)	Natural-language string	Natural-language reflection + rollout score	[1]
DSPy	2023	Compound-system parameters (instruction + demos per module)	Bootstrap self-generation + random search + coordinate ascent	Modular DSL (typed signatures)	Scalar end-to-end metric	[22]
MIPRO / MIPROv2	2024	Compound-system parameters (instructions + demos)	LLM proposals + Bayesian optimization (TPE surrogate)	Modular DSL	Scalar metric via a mini-batch surrogate	[33]
TextGrad	2024	Compound-system variables (prompts, code, etc.)	LLM-as-optimizer, textual-gradient descent	Computational graph of variables	Per-node natural-language critique	[58]
Trace / OptoPrime	2024	Compound-system variables	LLM-as-optimizer over serialized trace subgraphs	Computational graph (DAG)	Rich trace feedback (DAG + output feedback)	[5]
AFlow	2024	Workflow / topology (code graph)	LLM-guided MCTS (soft mixed-probability selection)	Executable code with operators	Execution score (+ cost)	[63]
GPTSwarm	2024	Workflow topology (edges) + node prompts	REINFORCE policy gradient over edge probabilities	Probability-weighted graph	Outcome reward (task metric)	[67]
DyLAN	2023	Agent-team membership + dynamic depth	Heuristic backward peer rating (Agent Importance Score)	Layered agents with a forward pass	Task metric + unsupervised peer ratings	[28]
MemAPO	2026	Prompt	Memory-guided reflective LLM optimization	Natural-language string	NL critique + scored memory of past edits	[26]
MASPOB	2026	Multi-agent prompts	UCB bandit over a GNN-encoded prompt space	Graph (GNN-encoded prompts)	Scalar bandit reward	[17]
TPGO	2026	Compound parameters + topology	Textual gradients + self-improving meta-optimization	Textual parameter graph	Per-node NL critique	[14]
Workflow-R1	2026	Workflow program	RL (group policy optimization over sub-sequences)	Generated workflow program	Outcome reward (spread across steps)	[23]
SWIFT	2026	Workflow topology	Amortized cross-query transfer	Graph / topology	Transferred execution reward	[8]

Table 2. Taxonomy of representative methods for the automated design of agentic systems, classified by optimization target, search strategy, representation, and feedback signal (part 2 of 2: RL-over-workflow methods, module/architecture search, and self-evolving systems; including 2026 entries).

Method	Year	Target	Search strategy	Representation	Feedback signal	Ref.
ScoreFlow	2025	Workflow program (generator weights)	Score-DPO (preference optimization with quantitative feedback)	Generated code	Scored numerical results in preference pairs	[49]
AgentSquare	2024	Agent modules (4-slot cell) + code	LLM-evolutionary (mutation + recombination, hill climbing)	Code + module sets with standardized I/O	In-context surrogate performance predictor	[42]
MaAS	2025	Agentic-architecture distribution (per query)	Cost-aware MC policy gradient + textual gradient	Probabilistic layered supernet	Reward minus token/API cost	[60]
Archon	2024	Inference-time layered-technique workflow	Bayesian optimization (Gaussian-process surrogate)	Typed layered DAG grammar	Held-out dev-set accuracy under a budget	[39]
MASS	2025	Topology + block/workflow prompts	Staged coordinate optimization (MIPRO + influence-pruned topology)	Discrete topology + natural-language prompts	Validation-set accuracy + influence proxy	[64]
Meta Agent Search (ADAS)	2024	Full code system	LLM-as-optimizer, open-ended evolutionary archive	Turing-complete code + natural-language idea	Scalar score + novelty + error-trace reflection	[18]
EvoAgent	2024	Agent modules (persona) → multi-agent topology	Evolutionary (population GA, LLM operators)	Natural-language personas / settings	Quality check with LLM-as-judge	[56]
STOP	2023	Full code (improver)	Recursive self-improvement (LLM, frozen weights)	Code	Downstream-utility score	[59]
Agent Symbolic Learning	2024	Modules + topology (prompts, tools, pipeline)	Textual-gradient backpropagation analogue	Modular pipeline DSL	Language loss / label-free LLM critique	[65]
Gödel Agent	2024	Full code system (self-referential)	LLM-as-optimizer hill climbing, recursive self-rewriting	Runtime code	LLM-estimated utility	[55]
Darwin Gödel Machine	2025	Full code system (self-improving)	Open-ended quality-diversity (archive + LLM mutation)	Agent code	Empirical benchmark (SWE-bench / Polyglot)	[62]
Learn-to-Configure	2026	Per-query configuration	Hierarchical RL policy (inference-time)	Discrete modular configuration	Outcome reward	[45]
EvoMAS	2026	Multi-agent system	Evolutionary MAS generation (LLM operators)	Natural-language agent specs / topology	Fitness (LLM-judge / score)	[19]
MOSS	2026	Full code system (self-rewriting)	Source-level recursive self-rewriting	Source code	Empirical benchmark	[3]

4. Automatic Prompt Optimization

At the most accessible level of the ADAS hierarchy lies *automatic prompt optimization* (APO): discovering, from data, the natural-language instruction that maximizes the effectiveness of a frozen LLM on a task. All methods treat a discrete, human-readable prompt as the optimization variable with fixed weights—a black-box, gradient-free search over the non-differentiable token space whose only meaningful neighbors are semantic rewrites. The unifying trick is to use *the LLM itself as the search*

 doi:10.20944/preprints202606.0238.v1

operator: it proposes, mutates, critiques, or recombines instructions, while a held-out set supplies a fitness signal. The methods differ by feedback signal, mutation operator, selection/scoring, and global search strategy.

APE [66] set the pattern, treating an instruction as a program optimized over a set of LLM-proposed candidates. *Forward mode* infers an instruction from input–output demonstrations; *reverse mode* uses infilling models to place it anywhere by filling in the blanks. Candidates are scored by execution accuracy or answer log-likelihood. An adaptive filter scores all candidates on a small subset, discards low scorers, and allocates a larger budget to the survivors; an optional iterative Monte Carlo search resamples high-scoring candidates. This propose–score–filter loop matched or exceeded humans on 19 of 24 tasks.

OPRO [53] generalizes this into an optimization-trajectory paradigm: a *meta-prompt* contains the task description, examples, and a trajectory of prior instructions with their scores sorted in ascending order. The LLM generates new instructions meant to surpass those seen so far, optimizing implicitly by pattern-matching over (solution, score) pairs—with no gradient or update rule. Each step samples a mini-batch, scores it on a subset, and appends the pairs. OPRO's optimizer LLM discovered the instruction "Take a deep breath and work on this problem step by step," which reached 80.2% on GSM8K, with PaLM 2-L serving as a separate scorer. Its distinguishing feature is a *scored memory*; its weaknesses are a context-bounded horizon and instability with small optimizer LLMs.

ProTeGi [35] makes the numerical analogy explicit. For a given prompt p_0, a mini-batch of errors feeds a meta-prompt that describes the deficiencies of prompt p_0 (a *textual gradient* g); an editing meta-prompt then edits p_0 "in the opposite semantic direction from g," and a paraphraser adds local Monte Carlo exploration. Candidates form a tree pruned by beam search. Selection is posed as best-arm identification, with UCB bandit sampling—making textual gradients budget-aware but dependent on the quality of the critic.

EvoPrompt [12] replaces reasoning with classical *evolutionary algorithms*, using the LLM as a variation operator that preserves coherence. The genetic (GA) variant selects parents by roulette wheel ($p_i \propto$ fitness), prompts the LLM to crossover then mutate, and truncates 2N to the best N. The differential-evolution (DE) variant mirrors $a + F(b - c)$ in language: the LLM identifies the spans where two prompts differ, mutates them, integrates with the current best, and performs crossover. A fixed population ($N \approx 10$) over T generations yields improvements on Big-Bench Hard, at a high call cost.

Promptbreeder [11] evolves *the operator itself* under a binary tournament GA, distinguishing task prompts from *mutation prompts* that are themselves evolved self-referentially. Nine operators span direct, estimation-of-distribution, hypermutation (evolving the mutation prompts), Lamarckian (back-deriving prompts from solutions), and crossover with context shuffling. The search distribution thus adapts, outperforming CoT and Plan-and-Solve—but it is computationally demanding.

PromptAgent [47] reformulates APO as planning in an MDP—states are prompts, actions are edits based on error feedback, and the reward is effectiveness—governed by MCTS (selection via UCT, expansion via reflection on errors, simulation, backpropagation). Unlike greedy descent or shallow populations, it does *lookahead*, reaching expert-level prompts on 12 tasks.

GEPA [1] performs *reflective prompt evolution* on the boundary between prompt and compound system: it mutates prompts but optimizes them with respect to full system trajectories rather than isolated input–output pairs. It samples trajectories, reflects on them in natural language to diagnose failures, and recombines complementary lessons along a Pareto front of attempts. Multi-agent prompt optimizers add structural priors: MASPOB [17] (2026) runs UCB bandits over a GNN-encoded prompt space to jointly optimize prompts across an agentic graph, so a single prompt's reward is informed by its position in the agent topology rather than scored in isolation. A second 2026 entry, MemAPO [26], augments reflective optimization (in the lineage of ProTeGi and PromptAgent) with a persistent memory of past edits and their outcomes, so that each reflection is conditioned on accumulated history

 doi:10.20944/preprints202606.0238.v1

rather than only the current mini-batch of errors—trading the context-bounded horizon of OPRO's scored memory for a durable store.

These systems share real limitations: expensive evaluation, overfitting to small scoring sets and to specific evaluators, a noisy 0–1 feedback signal, and poor transfer between models—which motivates the higher levels of ADAS that optimize workflows and full code systems.

5. Compiler-Style Optimization of Compound LLM Systems

A distinct line of research within ADAS regards a multi-step LM application as a *program*—a composition of LM calls, tools, control flow, and intermediate variables—whose tunable parts are jointly optimized with respect to an end-to-end objective. A compound AI system is largely a non-differentiable computational graph whose "weights" are prompts, instructions, few-shot demonstrations, code, and hyperparameters, and that is "learned" by having an LLM act as an optimizer or by Bayesian search over discrete configurations. Three bodies of work crystallize this view: DSPy and its teleprompters [22,33], TextGrad [58], and Trace/OptoPrime [5]. All share the abstraction that *the program is the search space*, yet they differ in representation, feedback signal, and search operator.

5.1. DSPy, MIPRO, and Bayesian Joint Optimization

DSPy recasts prompt engineering as programming [22]: a *signature* declares a typed subtask (`question -> answer`), and a *module* (`Predict`, `ChainOfThought`, `ReAct`) realizes it via an LM call, composing into "text transformation graphs" [22]. Each module's prompt is parameterized by a natural-language instruction, few-shot demonstrations, and signature formatting; a *teleprompter* acts as a compiler that "will optimize any DSPy pipeline to maximize a given metric" [22]. The core operator is *bootstrapping*: `BootstrapFewShot` runs the program over training inputs, records execution traces, and keeps the intermediate input/output pair from passes that pass the metric as self-generated demonstrations—labeling its own intermediate steps purely on the basis of supervision at the final output. `BootstrapFewShotWithRandomSearch` wraps this in a random search over demonstration sets; `COPRO`, in turn, optimizes the instruction text by coordinate ascent. DSPy optimizes prompts and demonstrations; its weight-tuning teleprompters belong to a separate strand of joint weight-and-scaffold optimization.

MIPRO unifies these, optimizing prompts for multi-stage pipelines "without access to module-level labels or gradients" [33]. It factorizes the joint string space, separately optimizing each module's instructions and demonstrations, through three stages: bootstrapping candidate demonstrations; a *grounded-proposal* stage in which a proposer LM drafts instructions from the program structure, dataset summaries, traces, and history [33]; and a discrete search that assigns each module an (instruction, demonstration-set) pair. The driving mechanism is Bayesian optimization—a Tree-structured Parzen Estimator over a "stochastic mini-batch evaluation function" [33]—which statistically resolves credit assignment, with a "meta-optimization procedure" [33] governing the proposer. MIPROv2 packages this Bootstrap→Propose→Search loop. The lineage reaches back to APE, which searched LM-proposed instructions to maximize a score [66], and to OPRO, which made the LLM an optimizer via a history of (solution, score) pairs [53].

5.2. TextGrad, Trace, and a Unifying View

TextGrad builds an automatic-differentiation engine over text [58], "backpropagat[ing] textual feedback provided by LLMs to improve individual components of a compound AI system." Modeled on PyTorch, a `Variable` holds text and tracks gradients; the forward pass builds a graph; `TextLoss` critiques the output; the backward pass propagates *textual gradients*—a per-edge natural-language critique, provided by the backward-engine LLM—and the *Textual Gradient Descent* optimizer rewrites each variable.

Trace regards *the whole execution trace* as the optimizer's input, arguing that traces are "akin to back-propagated gradients in AutoDiff" [5]. It defines **OPTO** (Optimization with Trace Oracle): a heterogeneous parameter space, a fixed context, and a trace oracle that returns a DAG with feedback.

 doi:10.20944/preprints202606.0238.v1

The **Trace** library turns workflows into OPTO via PyTorch-like `node`/`bundle` primitives; **OptoPrime** serializes the trace subgraph into a pseudocode-style report and instructs GPT-4 to reason and then propose updates—a *pseudo-gradient* over the entire graph. Unlike TextGrad's per-edge-decomposed descent, OptoPrime reasons globally in a single call.

These frameworks span three axes. *Representation:* DSPy/MIPRO optimize the instructions and demonstrations of a fixed graph of typed modules; TextGrad/Trace treat arbitrary variables as the parameters of a recorded graph. *Feedback signal:* a scalar metric with surrogate modeling versus natural-language critiques consumed by an LLM optimizer. *Operator:* self-bootstrapping and Bayesian TPE search versus descent in which an LLM is the optimizer, whether per-edge or over the whole graph. All four approaches optimize the parameters of a fixed topology—they do not invent new modules or control flow, unlike ADAS methods that mutate structure. LLM-as-optimizer approaches inherit vague, contradictory, or hallucinated gradients with no convergence guarantee; credit assignment is statistically noisy or reasoning-bounded, and evaluation is expensive. Still, the program view is today the backbone of ADAS, treating prompts, demonstrations, and code as uniformly optimizable parameters of a single end-to-end objective. A 2026 extension, TPGO [14], pushes the textual-gradient view beyond fixed-topology parameters: it threads textual gradients through a graph of textual parameters that spans both the compound system and its topology, and adds a self-improving meta-optimization loop that refines the optimizer itself—lifting Promptbreeder's self-referential idea (§4) to the compound-system level.

6. Automated Optimization of Workflow and Topology

Much of the ADAS field targets not a single agent but the *structure of collaboration* among many LLM calls: a graph describing who calls whom, with which prompts and with what aggregation logic. These methods share one abstraction—an agentic system is a computational graph (or program) whose nodes are LLM or tool calls and whose edges encode information flow—yet they differ in (i) what they optimize, (ii) how they represent the search space, and (iii) which optimizer they use (1).

GPTSwarm [67] is the canonical formalization of the "agents as optimizable graphs" idea, in which "nodes implement functions to process multimodal data or query LLMs, while edges describe the flow of information" [67]. *Node optimization* refines each node's prompt via an improver that maps the history of (input, output) pairs and the current prompt into a revised version. *Edge optimization* parameterizes candidate edges as $\theta \in [0,1]^d$, inducing a distribution over directed acyclic graphs, and solves $\arg\max_\theta \mathbb{E}_{G\sim D_\theta}[u_\tau(G)]$ using a REINFORCE estimator over sampled graphs. The feedback signal is a scalar measure. It is lightweight, but the candidate edge set is fixed and REINFORCE is high-variance, so the method tunes connectivity within a given inventory of agents rather than inventing new node types.

DyLAN [28] addresses team composition, objecting that "existing approaches are limited by using a fixed number of agents and static communication structures" [28]. It unrolls a dialogue as a temporal feed-forward network of agents arranged in layers across rounds, $\pi_\ell(O) = p(O \mid A_{1:\ell-1})$, with an LLM ranker performing early stopping. Its unsupervised *Agent Importance Score* aggregates mutual ratings backward through the layered graph to score agents, then removes the low-scoring ones—a single-pass, label-free pruning of team members that optimizes node membership and dynamic depth.

6.1. Methods Based on Code Search and Generators

AFlow [63] pushes the representation to arbitrary code, "reformulating workflow optimization as a search problem over code-represented workflows, where LLM-invoking nodes are connected by edges" [63]. It predefines reusable *operators* (Generate, Format, Review, Revise, Ensemble, Test, Programmer) and searches via a four-step MCTS loop: a soft *selection* with mixed probabilities; an *expansion* in which an optimizer LLM reads the workflow and a tree-structured *experience* log and emits modified code; an *evaluation* that executes candidates on a validation set for a mean-and-variance-based reward; and *backpropagation* into the experience tree. It surpasses the best prior automated workflow

 doi:10.20944/preprints202606.0238.v1

baseline by 5.7% and matches GPT-4o at just 4.55% of its inference cost, though it is expensive because of repeated execution.

Another family of methods trains a *generator*. **ScoreFlow** [49] criticizes discrete optimizers as "inflexible due to representation constraints" [49] and uses continuous gradient-based optimization: it samples workflows, executes them for numerical scores, builds preference pairs, and fine-tunes via *Score-DPO*, "a novel variant of direct preference optimization that accounts for quantitative feedback" [49]—incorporating the magnitude of the score difference, so that larger quality gaps yield stronger gradients.

Two 2026 methods continue this generator line. Workflow-R1 [23] casts workflow construction itself as reinforcement learning, applying group policy optimization over sub-sequences of the generated program so that credit is spread across construction steps rather than attributed only to the final outcome—a partial answer to the outcome-only-reward weakness shared by GPTSwarm and ScoreFlow. SWIFT [8] attacks the dominant cost of these methods, the per-query search: instead of searching a topology afresh for every query, it amortizes the search by transferring learned workflow structure across queries and tasks, trading some per-instance optimality for a large reduction in search cost.

Across all these methods, the *representation* spans probability-weighted edges (GPTSwarm), team membership (DyLAN), and executable code (AFlow, ScoreFlow); the *optimizers* range from REINFORCE through LLM-guided MCTS to Score-DPO; and the *feedback signal* is almost always an execution-based scalar, sometimes supplemented with peer ratings (DyLAN). Code-search methods are the most expressive but also the most expensive, since every candidate must be executed on the validation set—the dominant cost and the risk of overfitting.

7. Search over Modular Agents and Architectures

7.1. From Open-Ended Code Search to Structured Architecture Search

The earliest formulation of ADAS treats an agent as an arbitrary program and lets a meta agent write new agents from scratch, collecting discoveries in an archive [18]. Because "programming languages are Turing-complete, this approach theoretically enables learning any possible agentic system" [18], it is maximally expressive but minimally controllable. A second wave deliberately *constrains* the design space and borrows neural architecture search (NAS). The classical NAS decomposition into search space, search strategy, and performance estimation [10] maps neatly onto agent design: AgentSquare [42], MaAS [60], Archon [39], and MASS [64] each factorize the program space to make it searchable and add a cheap surrogate to make search affordable. **AgentSwift** [25] scales search with a learned value model that guides a hierarchical MCTS over a joint space of workflows and components, while Learn-to-Configure (Taparia et al. [45], 2026) learns a hierarchical RL policy that selects a *query-specific* configuration at inference time, generalizing MaAS's per-query idea into an explicitly amortized, learned policy rather than a per-query search.

AgentSquare is the analogue of cell-based NAS, but on the agent side [42]. It abstracts designs into four standardized modules—planning, reasoning, tool use, and memory—with a unified input-output contract, so that any module can be swapped for another module of the same type. The search interleaves *module evolution* (an LLM mutates one module at the code level) and *module recombination* (another LLM assembles existing modules from the task description and prior results), climbing uphill from the best agent so far. For performance estimation it learns "a performance predictor that uses in-context surrogate models to skip unpromising agent designs" [42], filtering proposals before expensive runs. It outperforms hand-crafted agents by 17.2% (relative to human-designed baselines) on six benchmarks. Its limitation: a fixed single-agent four-slot flow precludes new control flow.

MaAS borrows the supernet idea from one-shot NAS, "optimizing an agentic supernet, a probabilistic and continuous distribution of agentic architectures" [60] instead of a single system. The layers carry distributions over *agentic operators* (chain of thought, debate, self-reflection, ensembling, early exit), and a system is a sampled path. A query-conditioned controller samples a

 doi:10.20944/preprints202606.0238.v1

query-specific subnet—cheap for easy queries, rich for hard ones—under the cost-aware objective $\min \mathbb{E}[-p(a \mid q, \pi, O) + \lambda C(G; q)]$. The optimization is hybrid: a Monte-Carlo policy gradient updates the distribution, while textual gradients update the operator prompts. It needs only 6–45% of the inference cost of the baselines while surpassing them by 0.54–11.82%. Limitations: a fixed operator vocabulary and number of layers, and high-variance policy gradients.

Archon targets inference-time architecture wrapping around frozen LLMs [39]: a typed layered DAG of generators, critics, rankers, fusers, verifiers, and unit-test generators/evaluators, with construction rules that prune valid configurations. It searches by Bayesian optimization with Gaussian-process surrogates over the hyperparameters, against held-out accuracy under a token budget, far more sample-efficiently than greedy or random search, outperforming o1, GPT-4o, and Claude-3.5 Sonnet by 15.1% on one benchmark. Its reach is the orchestration of fixed models, not their internal cognitive modules.

MASS jointly optimizes prompts and topology, interleaving "from local to global, from prompts to topologies, in three stages: 1) block-level (local) prompt optimization; 2) workflow topology optimization; 3) workflow-level (global) prompt optimization" [64]. Stage 1 uses MIPRO; stage 2 searches five building blocks (aggregation, reflection, debate, summarization, tool use), pruning them by an *incremental-influence* metric; stage 3 re-optimizes prompts globally. Its limitation is staged greedy path-dependence and a heuristic influence surrogate.

Across the entire NAS triad, the deepest division is the one between a *single best architecture* (AgentSquare, Archon, MASS) and MaAS's *distribution* that emits, for each query, a cost-matched architecture. All four approaches bound expressiveness with a hand-designed vocabulary—unlike the open-ended code search of [18]—and transfer across tasks remains the weakest link.

8. Evolutionary and Self-Evolving Agents

An increasingly prominent branch of ADAS conceives of building agents as open-ended iteration: agents, or the meta-systems that produce them, are gradually transformed by mutation, recombination, and selection. It draws on quality-diversity / open-endedness (MAP-Elites [31]) and recursive self-improvement (Schmidhuber's Gödel machine [41]). What links these methods is that *the search operator itself is an LLM*: classical genetic operators become natural-language- or code-level transformations, moving from constrained (prompts, personas) to radical (code self-rewriting).

8.1. Evolving Prompts, Populations, and Workflows

Promptbreeder [11] optimizes *prompts* as the unit of variation: a population of task prompts is scored on a training set, an LLM mutates variants, and tournament selection keeps the high-fitness members. Its signature move is *self-referentiality*—the mutation operators are themselves co-evolved "mutation prompts": "Promptbreeder not only improves task-prompts, but it also improves the mutation-prompts that improve those task-prompts" [11].

EvoAgent [56] extends this to full *agents* and the generation of multi-agent systems, motivated by the fact that systems "rely heavily on human-designed frameworks, which significantly limits functional scope and scalability" [56]. Its genome is the set of an agent's *settings* (role, persona, prompt, skills): an LLM recombines and mutates parent settings into offspring, an LLM quality-check acting as a fitness filter scores inheritance and distinctiveness, and the survivors seed the next generation before being assembled into a multi-agent system. The genome is restricted to text, so the topology and code remain unchanged—within an existing scaffold it diversifies roles rather than inventing structure. EvoMAS [19] (2026) carries this evolutionary recipe up to the generation of entire multi-agent systems, evolving populations of agent teams—their roles, prompts, and interconnections—rather than single agents, and marks the maturation of evolutionary multi-agent-system design. AFlow [63] instead evolves *workflow topology*, searching code-represented workflows (LLM-invoking nodes connected by edges) via Monte Carlo tree search with operators and execution feedback. It is precisely AFlow's success that shows code representation unlocks structural change that EvoAgent cannot reach.

 doi:10.20944/preprints202606.0238.v1

8.2. Symbolic Learning and Recursive Self-Modification

Agent Symbolic Learning [65] self-evolves agents after deployment without retraining weights, modeling them as symbolic networks whose learnable "weights" are prompts, tools, and their composition, optimized by mimicking backpropagation. The forward pass stores a trajectory; a prompt-based *language loss* produces a label-free critique with an associated score; *language gradients*—"textual analyses and reflections used to update each component of the agent with respect to the language loss" [65]—are propagated backward so that upstream nodes receive downstream requirements. Three symbolic optimizers apply updates: PromptOptimizer edits descriptions/examples/format, ToolOptimizer improves/deletes/creates tools, and PipelineOptimizer restructures the graph by atomic node addition/deletion/reordering. This is a rare case of optimizing *tool synthesis* rather than mere tool use—which, in an embodied setting, was also exemplified by Voyager's library of self-written reusable skills [46]—but its "gradient" is an uncalibrated critic with no stability guarantees.

A key precursor is **STOP** (Self-Taught Optimizer) [59]: an initial *improver* uses an LLM to optimize a solution toward downstream utility, then improves *itself*; with frozen weights it improves the scaffold rather than performing full recursive self-improvement. The most ambitious methods rewrite *their own code*. Gödel Agent [55] "leverages LLMs to dynamically modify its own logic and behavior, guided solely by high-level objectives through prompting" [55] via introspection and runtime monkey-patching; unlike ADAS [18], it "recursively updates both the policy π and the meta-learning algorithm I" [55], sacrificing provable optimality for noisy hill-climbing in which a single bad rewrite can corrupt the agent. The Darwin Gödel Machine (DGM) [62] removes that fragility: "Inspired by Darwinian evolution and open-ended exploration, DGM maintains an archive of generated coding agents" [62]. It samples a parent, prompts for an "interesting" variant, empirically validates it on SWE-bench/Polyglot, and re-archives it—a branching into a tree that echoes MAP-Elites [31]. On the same code axis, but outside the design of agents themselves, lies AlphaEvolve [32]: an evolutionary coding agent that couples LLMs with automated evaluation to search the space of algorithms, reaching results such as faster matrix multiplication—proof that code-level search reaches algorithmic discovery, not just the assembly of agents. Empirical validation relaxes the Gödel machine's intractable proof requirement; the archive prevents lineage collapse, at a high computational cost. Extending this line, MOSS [3] (2026) pushes self-modification down to the source level, rewriting its own source code rather than patching behavior at runtime—in the lineage of STOP, Gödel Agent, and DGM, and at the most expressive and most hazardous end of the target axis.

8.3. Synthesis

A gradient of expressiveness versus controllability runs through all the methods: text-evolving Promptbreeder/EvoAgent are safe but limited; topology-evolving symbolic learning/AFlow are more expressive with credit assignment; code-evolving Gödel Agent/DGM are the most expressive and the most dangerous. A recent survey consolidates the field by *what, when, how, where* to evolve [2]. We omit from consideration methods that update model *weights* alongside the scaffold—e.g. the weight-tuning DSPy teleprompters and the GRPO-style RL against which GEPA is compared [1,22]—focusing on frozen-model scaffold search. Recurring open problems: the absence of stability/improvement guarantees with LLM operators, the reliability of self-generated fitness, the computational cost of archive search, and the safety of agents that pursue their own goals.

9. Reinforcement Learning and Reward-Driven Optimization

A distinct set of methods views scaffold design not as asking the LLM to *write* a better system (the LLM-as-optimizer direction—Meta Agent Search [18], AFlow's MCTS-over-code [63]) but as *stochastic optimization*: a parameterized distribution over scaffolds is pulled toward high-reward regions via policy gradients or preference learning (DPO [37]). This is closer to NAS than to prompt design and differs from agentic RL/RLHF that updates weights (e.g. MAPoRL [34]), which is outside our scope; scaffold methods keep the executor LLMs frozen and optimize only the structure around them.

 doi:10.20944/preprints202606.0238.v1

Two methods mentioned earlier form a matched pair. **GPTSwarm** [67] represents the *single* case: its edge optimization, whose objective was given earlier, is solved by a REINFORCE estimator over sampled graphs—high-variance and without a baseline (HumanEval pass@1 from 76% to 88%, one benchmark, one backbone). **MaAS** [60] represents the *distributional* case: its cost-aware objective, given earlier, is optimized by an empirical-Bayes Monte-Carlo procedure whose normalized importance weights are exactly the variance-reduction baseline that GPTSwarm lacks. Preference-based methods replace the policy gradient with ranking: Score-DPO within **ScoreFlow** [49] incorporates *cardinal* scores into the DPO loss so that reliable samples dominate (an 8B-parameter generator outperforms GPT-4o-mini-based optimizers). **MASS** [64] interleaves prompt and topology search and shows that prompt optimization must precede topology search.

A shared weakness is *outcome-level* reward, with no credit assignment per edge, operator, or agent. Lightman et al. [27] showed that on MATH process supervision outperforms outcome supervision, yet no scaffold-search method adopts a learned process-reward model. Coupling learned process rewards with variance-reduced policy gradients over frozen-model topologies is the clearest open research direction.

10. Search Methods: A Cross-Cutting View

10.1. A Unifying Optimization View

Despite the profusion of named systems, ADAS search reduces to approximately solving $\max_{c \in \mathcal{C}} \mathbb{E}_{(x,y)\sim\mathcal{D}}[u(\Phi_c(x), y)] - \lambda\, \mathbb{E}_x[\text{cost}(\Phi_c, x)]$, where a configuration c induces a system Φ_c, u is task utility, and λ penalizes token count/latency/calls. This mirrors neural architecture search (NAS), whose three pillars—search space, search strategy, and performance estimation—structure ADAS as well [18]. The differences are: the space is symbolic and often Turing-complete; the estimator is an expensive stochastic LLM rollout; and the operator itself is often an LLM.

Six operator families recur—LLM-as-optimizer (OPRO), textual gradients (ProTeGi, TextGrad, Trace, Agent Symbolic Learning), evolutionary and archive-based (EvoPrompt, Promptbreeder, Meta Agent Search, DGM), MCTS (PromptAgent, AFlow), surrogate/Bayesian (MIPRO, AgentSquare, Archon), and RL/DPO (GPTSwarm, ScoreFlow, MaAS)—elaborated above and shown in Tables~1–2.

10.2. Cross-Cutting Tensions

Beyond the two structural tensions introduced in the framework—expressiveness versus searchability, and feedback richness versus credit assignment—the cross-section of methods reveals two more. Exploration versus exploitation: novelty in the archive, bandit approaches in ProTeGi, MCTS in PromptAgent/AFlow, and policy entropy in GPTSwarm/MaAS. Meta-optimization, that is, optimizing the search operator itself (Promptbreeder, MIPRO). Only a minority of systems (AFlow, MaAS, ScoreFlow, Archon) treat λ cost as a genuine objective, making surrogate-guided, cost-aware estimation—a Pareto formulation inherited from multi-objective NAS—the most significant research frontier.

11. Evaluation, Benchmarks, and Metrics

ADAS inverts the usual relationship between an agent and a benchmark: the benchmark *is* the optimization signal. A meta-optimizer maximizes the validation-set score, so the choice of benchmark, the data-split protocol, and the very definition of "score" become the critical decisions on which everything rests.

11.1. The Standard Set of Benchmarks

A small, recurring set of benchmarks—almost none of which was designed for agent search—is repurposed into fitness functions. The reasoning and question-answering group includes GSM8K [7] and MMLU [15]; the coding group uses HumanEval [4] and the realistic SWE-bench [20]; the agentic and tool-use group includes GAIA [30] and benchmarks for web and tool work. The split matters: ADAS and AFlow rely on near-saturated reasoning and coding sets, with little room for progress and

 doi:10.20944/preprints202606.0238.v1

a high risk of contamination; AgentSquare and GPTSwarm reach for unsaturated agentic sets, where architecture does matter, but stochasticity and scoring ambiguity dominate.

11.2. Methods and Their Headline Numbers

Evaluating the surveyed methods reveals more shared patterns than isolated figures. Individual gains—ADAS's transfer of agents found on ARC, AFlow's work on the cost Pareto frontier, AgentSquare's surpassing of human-designed systems, GPTSwarm's robustness on an adversarial variant of MMLU, and MaAS's cost-aware supernet—are elaborated in earlier sections and summarized in Tables~1–2; here we read them through shared indicators rather than again method by method.

Four families of indicators recur: (i) *cost* in tokens/dollars; (ii) the *Pareto frontier* / inference compute; (iii) *transferability/generalization*; and (iv) *search cost* (AgentSquare's surrogate, AFlow's pruning in MCTS), which ADAS's brute-force archive-and-evaluate loop ignores. Because evaluation protocols differ, such percentages are not comparable across the rows of Tables~1–2.

The sharpest critique, "AI Agents That Matter" [21], argues that evaluation focused solely on accuracy produces agents that are "needlessly complex and costly" and "brittle" because of shortcuts, proposing the joint optimization of cost and accuracy and an Agentic Benchmark Checklist—a framework that MaAS adopts. Concrete hazards recur:

- *contamination*: GSM8K, MMLU, and HumanEval are documented to be contaminated (~13.8% rates for MMLU [52]), and decontamination noticeably lowers measured accuracy;
- *weak baselines*: by our assessment, none of the surveyed methods reports a direct comparison with the strongest hand-designed multi-agent systems (MetaGPT, ChatDev, AutoGen [16,36,51]), despite the field's claim that "learned beats hand-designed";
- *lack of held-out sets*: optimizing *on* the validation split conflates search with generalization;
- *variance*: GPTSwarm reports bootstrap confidence intervals, yet many studies omit them;
- *reproducibility*: refreshed benchmarks are not yet widely adopted.

The open question is: do the discovered systems encode reusable reasoning structure, or merely a scaffold tailored to the benchmark?

12. Recent Developments (2026)

The first months of 2026 brought a surge of ADAS work along every axis—more than fifty papers in that window alone. Rather than wall these off as a separate category, we have placed the most relevant within the axis sections above and listed them in Tables~1–2: MemAPO and MASPOB under prompt optimization (§4), TPGO under compound-system optimization (§5), Workflow-R1 and SWIFT under workflow search (§6), Learn-to-Configure under modular architecture search (§7), and EvoMAS and MOSS under self-evolving agents (§8). Read together, two trends distinguish the 2026 work from that of 2024–2025.

First, *co-evolution*: prompts and topology are increasingly optimized jointly rather than each in isolation. MASPOB couples prompt optimization to the agent graph, and TPGO threads textual gradients through a shared graph of compound parameters and topology while meta-optimizing its own optimizer—both reflecting MASS's earlier lesson that prompt and topology search interact (§7).

Second, *amortization and transfer*: a decisive turn away from per-query search, whose cost is the dominant practical limit (§11). SWIFT transfers learned workflow structure across queries, and Learn-to-Configure replaces per-query search with a learned hierarchical policy that emits a configuration at inference time. Alongside these, Workflow-R1 brings reinforcement learning to workflow construction and EvoMAS and MOSS extend evolutionary and self-rewriting methods to whole multi-agent systems and to source-level self-modification, respectively. The direction of travel is clear: the field is moving from searching a system afresh for each input toward learning, once, a generator or policy that produces good systems cheaply.

 doi:10.20944/preprints202606.0238.v1

13. Challenges, Limitations, Safety, and Directions for Future Work

ADAS has matured exceptionally fast—from Meta Agent Search, which programs new agents as code [18], to recursively self-improving systems such as Gödel Agent [55] and the Darwin Gödel Machine (DGM) [62]—and in doing so has outpaced our understanding of when these systems work, how much they cost, and whether they are safe.

13.1. Cost, Evaluation, Generalization, Reproducibility

ADAS inherits NAS's pathology: expensive candidate evaluation in the inner loop makes the outer loop expensive [10], except that the economics are now measured in tokens rather than GPU resources. Meta Agent Search executes each candidate over the validation set through many LLM calls [18]; DGM multiplies this further because it scores each archived agent on benchmarks via dozens of long-context calls, with improvements (SWE-bench 20.0% $\rightarrow$ 50.0%) requiring compute that few research groups can afford [62]. The response is *cost-aware* search, embodied by AFlow [63]. The real objective is multi-criteria (accuracy, latency, cost, tokens).

A feedback signal is only as good as the evaluator that produces it. In the absence of truly correct labels, ADAS resorts to LLM-as-judge, which suffers from position bias and self-preference. When such a judge becomes the optimization target, a noisy fitness measure degenerates into *reward hacking*—this is Goodhart's law, by which over-optimizing a proxy measure undermines true performance. DGM gave a concrete example: when instructed to reduce hallucinations in tool use, it "removed the markers" [62] that the detector relied on, thereby faking high scores.

ADAS additionally overfits to benchmarks in two ways. The base model may be contaminated: GSM1K, a freshly collected version of GSM8K, showed accuracy drops of up to 8% associated with verbatim reproduction [61]. Moreover, the search itself may discover benchmark-specific tricks; Meta Agent Search uses Chollet's contamination-resistant ARC-AGI benchmark and claims transfer across domains and models [18], but there is no agentic benchmark with controlled contamination in the GSM1K style. Reproducibility is doubly hampered: the outer search is stochastic, and even temperature-zero decoding is nondeterministic—He et al. attribute this to non-associative floating-point arithmetic in GPU kernels, where changes of batch or GPU shift accuracy by about 9% [13], so the "winner" from the archive may be mere noise.

13.2. Safety, Interpretability, Co-Design, Theory

The deepest concern is recursive self-improvement. Schmidhuber's Gödel machine required *provably* useful modifications [41]; modern systems replace proof with empirical benchmark checking [62], and it is precisely that empirical-checking loop that lets an agent rewrite its own oversight. Gödel Agent goes a step further: by removing human-designed constraints and deferring to high-level objectives [55], the attack surface is maximized. Clune's AI-GA manifesto warned of these safety concerns at the very outset [6]. When safety arguments grounded in a fixed objective fail, the first line of defense is operational mitigations: sandboxing, human-supervised mutations, and verifier-based fitness rather than self-scoring [65].

Interpretability depends on the representation: code (Meta Agent Search, AFlow, DGM) is nominally readable but gradually accumulates opaque control flow; GPTSwarm's graphs can be monitored, but they are driven by fitness [67].

Joint optimization beckons enticingly—TextGrad's textual gradients [58], DSPy optimizing prompts and demonstrations [22], and AlphaEvolve, which found an algorithm to multiply 4$\times$4 complex matrices with 48 multiplications [32]—and all point toward closing a loop in which ADAS improves the very model that powers ADAS. ADAS remains almost entirely empirical: there is no learnability theory, no convergence analysis, and no characterization of when an LLM-as-optimizer beats random search [24].

ADAS sits at the intersection of surveys on self-evolution [2], multi-agent systems, prompt engineering [24], and NAS [10], and what sets it apart is its Turing-complete, code-level search space

 doi:10.20944/preprints202606.0238.v1

and the failure modes inherent to recursive self-improvement. A parallel survey by Yue et al. [57] maps the workflow-optimization part along a static-versus-dynamic axis, complementing our system-wide view.

14. Related Surveys and Conclusion

Several related surveys overlap with ours but differ in their lens. Elsken et al. survey neural architecture search (NAS) along the axes of search space, strategy, and estimation [10]; we adopt that triad as a backbone but elaborate it into four agent-specific axes and apply it to scaffolds rather than networks. A recent survey of self-evolving agents organizes the field around when, how, and where agents adapt, conceiving of self-evolution as bridging static foundation models and agentic systems capable of lifelong learning [2]; it centers adaptation at deployment time, whereas we center search at design time and its optimization theory. Kapoor et al. criticize benchmark-based agent evaluation for ignoring cost and rewarding overfitted complexity [21], and we fold that perspective into our analysis of evaluation rather than into the taxonomy. Our distinctive contribution is the four-axis decomposition, which places prompt-, compound-system-, workflow-, modular-, and code-level methods into a single comparative space and exposes the tensions between expressiveness and searchability, and between feedback signal and credit assignment, that pervade all these methods.

ADAS views the assembly of foundation models into agentic systems as a search problem. This four-axis framework shows that many named methods are points in one shared space, not mutually unrelated systems. The trajectory is legible: a climb up the ladder of optimization targets from prompts through compound programs and topologies to modular cells and self-writing code, governed by search tools borrowed from NAS, evolution, MCTS, Bayesian optimization, and reinforcement learning [10]. Progress is real but bounded by two tensions. Expressiveness enables structural novelty but at the cost of searchability and overfitting; feedback richness localizes credit but rests on hallucinated, unguaranteed signals with no process-level credit assignment. Evaluation further amplifies these problems: contaminated benchmarks, weak baselines, nondeterministic inner loops, and reward hacking make published gains hard to credit [21]. The most important frontier is cost-aware, contamination-resistant, transfer-verified evaluation; the deepest open problem is safety: as self-improvement becomes ever more open-ended, so grows the gap between what these systems can do and what we can verify [6].

References

1. Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.
2. Huan ang Gao, Jiayi Geng, Wenyue Hua, et al. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*, 2025.
3. Qianshu Cai, Yonggang Zhang, Xianzhang Jia, et al. Moss: Self-evolution through source-level rewriting in autonomous agent systems. *arXiv preprint arXiv:2605.22794*, 2026.
4. Mark Chen, Jerry Tworek, Heewoo Jun, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
5. Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the next autodiff: Generative optimization with rich feedback, execution traces, and llms. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.
6. Jeff Clune. Ai-gas: Ai-generating algorithms, an alternate paradigm for producing general artificial intelligence. *arXiv preprint arXiv:1905.10985*, 2019.
7. Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.
8. Shiyi Du, Jiayuan Liu, Weihua Du, et al. Why search when you can transfer? amortized agentic workflow design from structural priors. *arXiv preprint arXiv:2604.25012*, 2026.
9. Yilun Du, Shuang Li, Antonio Torralba, et al. Improving factuality and reasoning in language models through multiagent debate. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *PMLR*, pages 11733–11763, 2024.

 doi:10.20944/preprints202606.0238.v1

10. Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.
11. Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, et al. Promptbreeder: Self-referential self-improvement via prompt evolution. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *PMLR*, 2024.
12. Qingyan Guo, Rui Wang, Junliang Guo, et al. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *International Conference on Learning Representations (ICLR)*, 2024.
13. Horace He and Thinking Machines Lab. Defeating nondeterminism in llm inference. Thinking Machines Lab: Connectionism (blog), 2025.
14. Shan He, Runze Wang, Zhuoyun Du, et al. Learning to evolve: A self-improving framework for multi-agent systems via textual parameter graph optimization. *arXiv preprint arXiv:2604.20714*, 2026.
15. Dan Hendrycks, Collin Burns, Steven Basart, et al. Measuring massive multitask language understanding. In *International Conference on Learning Representations (ICLR)*, 2021.
16. Sirui Hong, Mingchen Zhuge, Jiaqi Chen, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations (ICLR)*, 2024.
17. Zhi Hong, Qian Zhang, Jiahang Sun, et al. Maspob: Bandit-based prompt optimization for multi-agent systems with graph neural networks. *arXiv preprint arXiv:2603.02630*, 2026.
18. Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations (ICLR)*, 2025.
19. Yuntong Hu, Yuting Zhang, Matthew Trager, et al. Evomas: Evolutionary generation of multi-agent systems. *arXiv preprint arXiv:2602.06511*, 2026.
20. Carlos E. Jimenez, John Yang, Alexander Wettig, et al. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations (ICLR)*, 2024.
21. Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, et al. Ai agents that matter. *Transactions on Machine Learning Research (TMLR)*, 2025.
22. Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, et al. Dspy: Compiling declarative language model calls into state-of-the-art pipelines. In *International Conference on Learning Representations (ICLR)*, 2024.
23. Mingze Kong, Zikun Qu, Zhongquan Zhou, et al. Workflow-r1: Group sub-sequence policy optimization for multi-turn workflow construction. *arXiv preprint arXiv:2602.01202*, 2026.
24. Wenwu Li, Xiangfeng Wang, Wenhao Li, and Bo Jin. A survey of automatic prompt engineering: An optimization perspective. *arXiv preprint arXiv:2502.11560*, 2025.
25. Yu Li, Lehui Li, Zhihao Wu, et al. Agentswift: Efficient llm agent design via value-guided hierarchical search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 31843–31851, 2026.
26. Guanbao Liang, Yuanchen Bei, Sheng Zhou, et al. Generalizable self-evolving memory for automatic prompt optimization. *arXiv preprint arXiv:2603.21520*, 2026.
27. Hunter Lightman, Vineet Kosaraju, Yura Burda, et al. Let's verify step by step. In *International Conference on Learning Representations (ICLR)*, 2024.
28. Zijun Liu, Yanzhe Zhang, Peng Li, et al. A dynamic llm-powered agent network for task-oriented agent collaboration. In *Conference on Language Modeling (COLM)*, 2024.
29. Kai Mei, Xi Zhu, Wujiang Xu, et al. Aios: Llm agent operating system. In *Conference on Language Modeling (COLM)*, 2025.
30. Grégoire Mialon, Clémentine Fourrier, Craig Swift, et al. Gaia: a benchmark for general ai assistants. In *International Conference on Learning Representations (ICLR)*, 2024.
31. Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.
32. Alexander Novikov, Ngan Vu, Marvin Eisenberger, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.
33. Krista Opsahl-Ong, Michael J. Ryan, Josh Purtell, et al. Optimizing instructions and demonstrations for multi-stage language model programs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2024.
34. Chanwoo Park, Seungju Han, Xingzhi Guo, et al. Maporl: Multi-agent post-co-training for collaborative large language models with reinforcement learning. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025.

 doi:10.20944/preprints202606.0238.v1

35. Reid Pryzant, Dan Iter, Jerry Li, et al. Automatic prompt optimization with "gradient descent" and beam search. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 7957–7968, 2023.
36. Chen Qian, Wei Liu, Hongzhang Liu, et al. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 15174–15186, 2024.
37. Rafael Rafailov, Archit Sharma, Eric Mitchell, et al. Direct preference optimization: Your language model is secretly a reward model. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
38. Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.
39. Jon Saad-Falcon, Adrian Gamarra Lafuente, Shlok Natarajan, et al. Archon: An architecture search framework for inference-time techniques. In *International Conference on Learning Representations (ICLR)*, 2025.
40. Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, et al. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
41. Jürgen Schmidhuber. Gödel machines: Fully self-referential optimal universal self-improvers. In Ben Goertzel and Cassio Pennachin, editors, *Artificial General Intelligence*, pages 199–226. Springer, 2007.
42. Yu Shang, Yu Li, Keyu Zhao, et al. Agentsquare: Automatic llm agent search in modular design space. In *International Conference on Learning Representations (ICLR)*, 2025.
43. Noah Shinn, Federico Cassano, Edward Berman, et al. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
44. Richard S. Sutton. The bitter lesson. Personal website (incompleteideas.net), 2019.
45. Aditya Taparia, Som Sagar, and Ransalu Senanayake. Learning to configure agentic ai systems. *arXiv preprint arXiv:2602.11574*, 2026.
46. Guanzhi Wang, Yuqi Xie, Yunfan Jiang, et al. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research (TMLR)*, 2024.
47. Xinyuan Wang, Chenxi Li, Zhen Wang, et al. Promptagent: Strategic planning with language models enables expert-level prompt optimization. In *International Conference on Learning Representations (ICLR)*, 2024.
48. Xuezhi Wang, Jason Wei, Dale Schuurmans, et al. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations (ICLR)*, 2023.
49. Yinjie Wang, Ling Yang, Guohao Li, et al. Scoreflow: Mastering llm agent workflows via score-based preference optimization. *arXiv preprint arXiv:2502.04306*, 2025.
50. Jason Wei, Xuezhi Wang, Dale Schuurmans, et al. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.
51. Qingyun Wu, Gagan Bansal, Jieyu Zhang, et al. Autogen: Enabling next-gen llm applications via multi-agent conversation. In *Conference on Language Modeling (COLM)*, 2024.
52. Cheng Xu, Shuhao Guan, Derek Greene, and M-Tahar Kechadi. Benchmark data contamination of large language models: A survey. *arXiv preprint arXiv:2406.04244*, 2024.
53. Chengrun Yang, Xuezhi Wang, Yifeng Lu, et al. Large language models as optimizers. In *International Conference on Learning Representations (ICLR)*, 2024.
54. Shunyu Yao, Jeffrey Zhao, Dian Yu, et al. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.
55. Xunjian Yin, Xinyi Wang, Liangming Pan, et al. Godel agent: A self-referential agent framework for recursive self-improvement. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 27890–27913, 2025.
56. Siyu Yuan, Kaitao Song, Jiangjie Chen, et al. Evoagent: Towards automatic multi-agent generation via evolutionary algorithms. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics (NAACL)*, pages 6192–6217, 2025.
57. Ling Yue, Kushal Raj Bhandari, Ching-Yun Ko, et al. From static templates to dynamic runtime graphs: A survey of workflow optimization for llm agents. *arXiv preprint arXiv:2603.22386*, 2026.
58. Mert Yuksekgonul, Federico Bianchi, Joseph Boen, et al. Optimizing generative ai by backpropagating language model feedback. *Nature*, 639:609–616, 2025.
59. Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (stop): Recursively self-improving code generation. In *Conference on Language Modeling (COLM)*, 2024.
60. Guibin Zhang, Luyang Niu, Junfeng Fang, et al. Multi-agent architecture search via agentic supernet. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025.

 doi:10.20944/preprints202606.0238.v1

61. Hugh Zhang, Jeff Da, Dean Lee, et al. A careful examination of large language model performance on grade school arithmetic. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2024.
62. Jenny Zhang, Shengran Hu, Cong Lu, et al. Darwin godel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025.
63. Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, et al. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations (ICLR)*, 2025.
64. Han Zhou, Xingchen Wan, Ruoxi Sun, et al. Multi-agent design: Optimizing agents with better prompts and topologies. In *International Conference on Learning Representations (ICLR)*, 2026.
65. Wangchunshu Zhou, Yixin Ou, Shengwei Ding, et al. Symbolic learning enables self-evolving agents. *AI Open*, 2025.
66. Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, et al. Large language models are human-level prompt engineers. In *International Conference on Learning Representations (ICLR)*, 2023.
67. Mingchen Zhuge, Wenyi Wang, Louis Kirsch, et al. Language agents as optimizable graphs. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *PMLR*, 2024.