Bilevel Autoresearch: When the Optimization Loop Optimizes Itself

Yaonan Qu[*]

Abstract

Autoresearch systems iterate an LLM-driven propose–train–evaluate loop to improve a target task, but their search mechanisms are fixed at design time. Human researchers introduce improvements such as multi-batch parallelism or persistent memory; the system itself cannot. We present Bilevel Autoresearch, a two-level framework in which an outer loop autonomously discovers and injects new search mechanisms into the inner loop. Level 1 runs the standard autoresearch loop; Level 1.5 analyzes the search trace and adjusts active parameters; Level 2 conducts a 4-round research session (Explore, Critique, Specify, Generate) and writes Python code that structurally modifies the inner loop at runtime via `importlib`. We evaluate three configurations on Karpathy's GPT pretraining benchmark (`val_bpb`) using the same DeepSeek model at every level. Adding Level 1.5 alone (Group B) yields no reliable gain over the pure inner loop (Group A): -0.007 ± 0.006 vs. -0.009 ± 0.001. Adding Level 2 (Group C) achieves -0.045 ± 0.030, a $5\times$ improvement over Group A. Across three independent repeats, Level 2 autonomously generates mechanisms from four distinct domains—combinatorial optimization, multi-armed bandits, Bayesian optimization, and design of experiments—without human specification. These results validate the principle that *autoresearch can research itself*: structural modification of the search loop, not merely parameter adjustment, is the decisive lever for large performance gains.

1 Introduction

Large language models have demonstrated a striking capacity for self-directed scientific iteration: given a task, an LLM can propose a change, execute an experiment, observe the outcome, and decide whether to keep or discard the change. When repeated, this propose–execute–evaluate loop constitutes a form of automated research [9]. Instantiated for neural network hyperparameter search, we call this loop *autoresearch*.

Despite its promise, autoresearch as currently practiced has a fundamental limitation: the search *mechanism* is fixed at design time. Every system in the literature uses a human-engineered architecture. Karpathy [9] introduced the single-track inner loop with a keep/discard acceptance rule. AutoResearchClaw [1] extended it with multi-batch parallel search. EvoScientist [4] added persistent experience memory across runs. A human designed each improvement by reading the prior system's code, identifying a bottleneck, and writing new code to address it. The systems themselves cannot perform this operation.

This raises a natural question: *can an outer loop perform that same design step autonomously?* Can the system read its own inner loop's code, identify structural bottlenecks, generate new Python code to address them, and inject that code at runtime—all without human involvement?

We answer this question affirmatively. We present **Bilevel Autoresearch**, a bilevel framework with three operational levels:

- **Level 1** (inner loop) runs the standard autoresearch cycle: an LLM proposes a configuration change, the model trains for a fixed budget, validation loss (`val_bpb`) is measured, and the change is kept or discarded.

[*]Independent Researcher, EdwardOptimization@gmail.com

- **Level 1.5** (outer loop) executes every five inner iterations. It analyzes the accumulated search trace, freezes parameters that have failed repeatedly, unfreezes under-explored parameters, and injects strategic guidance text into the next inner cycle.
- **Level 2** (mechanism research) executes every two outer cycles. It conducts a structured 4-round research session—Explore, Critique, Specify, Generate—producing Python code that implements a novel search mechanism. The code is patched into the inner loop's runner module via `importlib` dynamic loading and validated before activation.

Crucially, all three levels use the same LLM (DeepSeek `deepseek-chat`). Any improvement in search effectiveness therefore comes from the bilevel *architecture*, not from access to a more capable model.

We evaluate three experimental groups on Karpathy's GPT pretraining benchmark: Group A (Level 1 only), Group B (Levels 1 and 1.5), and Group C (all three levels). The primary metric is `val_bpb` improvement within a 30-iteration budget using a 300-second training window per iteration, matching the original benchmark specification. All groups start from the same baseline configuration and use identical GPU hardware (RTX 5090).

Contributions.

1. We formalize autoresearch as a bilevel optimization problem in which the upper level optimizes the search *mechanism* (runner code) and the lower level optimizes the task (training configuration), and demonstrate that the upper level can be solved by LLM code generation.
2. We implement Level 2 mechanism research via a 4-round LLM dialogue and runtime code injection, establishing a concrete realization of a self-modifying autoresearch system.
3. We report a controlled ablation showing that Level 2 produces a $5\times$ improvement over the Level-1 baseline (-0.045 ± 0.030 vs. -0.009 ± 0.001), with Level 2 autonomously generating mechanisms drawn from four distinct algorithmic domains.
4. We identify why Level 2 succeeds: the generated mechanisms (Tabu Search, Multi-Scale Bandit, Orthogonal Exploration) forced exploration of `TOTAL_BATCH_SIZE` *reduction*, a direction that the LLM's default search path systematically avoids due to an implicit bias toward larger batches.

Section 2 reviews related work. Section 3 describes the framework in detail. Section 4 presents ablation results. Section 5 interprets hypotheses, failure modes, and limitations. Section 6 concludes.

2 Related Work

2.1 Autoresearch and LLM-Driven Optimization

Karpathy [9] introduced the paradigmatic autoresearch loop for neural network hyperparameter search: an LLM reads a training script, proposes a configuration change, executes training for a fixed budget, measures validation loss, and accepts or rejects the change. Iterated, this constitutes a form of LLM-guided hill climbing in configuration space, where the LLM's world knowledge serves as an implicit prior over promising changes and training outcomes provide gradient-free feedback.

AutoResearchClaw [1] extends this framework with multi-batch parallelism: several candidate configurations are evaluated simultaneously, and the best is promoted. This increases the effective branching factor of search without altering the underlying acceptance mechanism.

EvoScientist [4] introduces persistent experience memory: lessons from prior runs are summarized and injected into future proposals, enabling cross-run learning. Both of these enhancements were designed by human researchers who inspected the prior system's code and identified architectural gaps. In all three systems, the structural decisions—when to accept, how to propose, what state to maintain—are made by human designers, not by the system itself.

2.2 Bilevel Optimization

Bilevel optimization [3, 16] studies problems of the form $\min_\phi F(\phi, \theta^*(\phi))$ subject to $\theta^*(\phi) \in \arg\min_\theta f(\theta, \phi)$, where an upper-level objective F depends on the optimal solution of a lower-level problem parameterized by ϕ. Applications include meta-learning [6], neural architecture search [13], and hyperparameter optimization [5]. In our setting the upper level optimizes the search *mechanism* ϕ (the runner code) and the lower level optimizes the task performance θ (the training configuration). The key departure from classical bilevel optimization is that ϕ is a program—a discrete artifact produced by code generation—rather than a real-valued parameter vector.

2.3 LLM-Based Code Generation for Research

AlphaCode [11] and Codex [2] demonstrated that LLMs can write functionally correct programs from natural language specifications. FunSearch [15] extended this to scientific discovery, using an LLM to iteratively generate and evaluate mathematical programs, finding new results in combinatorics. Most directly related is the line of work on LLM-driven algorithm design [12, 10], in which LLMs propose novel algorithmic variants that are then evaluated on benchmark tasks. Our Level 2 agent applies the same code generation capacity to a different target: rather than generating task-level programs, it generates *search mechanism code* that is injected into the inner loop at runtime.

2.4 Meta-Learning and Algorithm Configuration

Meta-learning [7] trains models to learn efficiently from few examples by optimizing across a distribution of tasks. Algorithm configuration [8] and algorithm selection [14] choose among candidate algorithms or parameter settings for a given problem instance. Portfolio methods [17] maintain a library of algorithms and select among them. Bilevel Autoresearch operates in a similar spirit—the outer loop selects or generates a search mechanism—but uses LLM code generation rather than gradient-based meta-optimization or a fixed portfolio.

2.5 Position of This Work

The key distinction between Bilevel Autoresearch and all prior work is the *target* of the outer loop. Level 1.5 is closest to existing outer loops (curriculum schedulers, adaptive configuration) in that it adjusts parameters of the existing mechanism. Level 2 is categorically different: it generates code that *replaces* the mechanism entirely. To our knowledge, Bilevel Autoresearch is the first system in which an autonomous outer loop writes and injects code that modifies the structural logic of the inner autoresearch loop at runtime, using the same model that runs the inner loop.

3 Methods

3.1 Framework Overview

Bilevel Autoresearch is organized as three nested operational levels. The inner loop (Level 1) optimizes the task; the outer loop (Level 1.5) optimizes the inner loop's search parameters;

Level 2 optimizes the inner loop's search *mechanism* by generating and injecting new code. All levels share the same DeepSeek `deepseek-chat` model; no stronger model is used at any outer level.

3.2 Level 1: Inner Autoresearch Loop

The inner loop implements the standard autoresearch cycle [9]. At each iteration t:

1. The LLM receives the current `train.py` (frozen at the best accepted configuration), the list of active editable parameters, any frozen parameters, and a strategic guidance string injected by Level 1.5.
2. The LLM proposes a change: a set of parameter name–value pairs and a one-sentence hypothesis.
3. The change is applied to a working copy of `train.py` and training runs for a fixed 300-second budget.
4. If the resulting `val_bpb` is lower than the current best, the change is *kept* (the best copy is updated); otherwise it is *discarded*.

The iteration budget is fixed at 30 per repeat. The initial configuration locks `DEPTH=8` and `ASPECT_RATIO=64` to prevent architecture-size changes; all other parameters (`LR`, `WEIGHT_DECAY`, `WINDOW_PATTERN`, `HEAD_DIM`, `TOTAL_BATCH_SIZE`, etc.) are editable.

3.3 Level 1.5: Outer Search-Strategy Loop

Level 1.5 executes every 5 inner iterations. It receives the full trace of proposals and outcomes and produces a `SearchConfig` update:

- **Freeze** parameters that have been proposed $\geq k$ times with zero net improvement (default $k = 3$).
- **Unfreeze** parameters that were frozen but have not been explored since the search moved to a new region.
- **Inject** a guidance string instructing the inner loop to prioritize under-explored parameters.

Level 1.5 can redirect search diversity but cannot change the proposal generation logic, the acceptance criterion, or the loop structure. These structural changes require Level 2.

3.4 Level 2: Mechanism Research and Code Injection

Level 2 executes every 2 outer cycles. It conducts a 4-round structured dialogue, each round making a single LLM call:

1. **Explore.** The LLM reads the full `runner.py` source and the search trace. It surveys mechanisms from adjacent fields (combinatorial optimization, online learning, design of experiments, Bayesian optimization) and proposes candidate improvements.
2. **Critique.** The LLM evaluates the candidate mechanisms against the observed failure mode (e.g., repetitive proposals, parameter fixation) and selects the most promising one.
3. **Specify.** The LLM writes a precise interface specification: class name, constructor arguments, key methods with signatures, and integration points in `runner.py`.

Algorithm 1 Bilevel Autoresearch (Group C configuration)

```
Input: baseline train.py, runner $\phi_0$, budgets $T$=30, $K$=5 (outer period), $M$=2 (L2 period)

$\theta \leftarrow$ baseline config;  $\phi \leftarrow \phi_0$;  $t \leftarrow 0$;  outer_cycle $\leftarrow 0$
while $t < T$ do
  for $k = 1$ to $K$ do
    proposal $\leftarrow$ LLMPropose($\theta$, $\phi$, guidance)
    $\theta' \leftarrow \theta \oplus$ proposal
    val $\leftarrow$ Train($\theta'$, budget=300s)
    if val < BestVal then
      $\theta \leftarrow \theta'$;  BestVal $\leftarrow$ val
    end if
    $t \leftarrow t + 1$
  end for
  guidance, frozen $\leftarrow$ Level1.5(trace)
  outer_cycle $\leftarrow$ outer_cycle +1
  if outer_cycle mod$M = 0$ then
    $\phi' \leftarrow$ Level2Research($\phi$, trace)
    if ValidateImport($\phi'$) then
      $\phi \leftarrow \phi'$
    else
      revert $\phi$
    end if
  end if
end while
return $\theta$
```

4. **Generate.** The LLM writes complete, runnable Python code implementing the specified mechanism, including any modifications to `runner.py` required to call it.

The generated code is applied by patching `runner.py` in place. Before activation, the patched module is loaded via `importlib.util.spec_from_file_location` with explicit `sys.modules` registration (required to support `@dataclass` decorators). If the import succeeds, the patched runner replaces the active runner for subsequent inner iterations. If the import fails (e.g., missing dependency), the original runner is restored from a pre-patch backup and the failure is logged. This validate-and-revert mechanism ensures that Level 2 failures are non-destructive.

3.5 Bilevel Optimization Formulation

Let ϕ denote the runner code (the mechanism) and θ the configuration of `train.py`. The bilevel objective is:

$$\min_{\phi} \; F(\phi,\, \theta^*(\phi)) \quad \text{s.t.} \quad \theta^*(\phi) \;=\; \arg\min_{\theta} \; f(\theta, \phi), \tag{1}$$

where $f(\theta, \phi) = \texttt{val\_bpb}(\theta)$ is the validation bits-per-byte of the model trained with configuration θ under runner mechanism ϕ, and $F(\phi, \theta^*) = \texttt{val\_bpb}(\theta^*)$ is the best achievable validation loss under ϕ. Level 2 approximately solves the upper problem by code generation; Level 1 approximately solves the lower problem by LLM-guided hill climbing.

3.6 Algorithm

Table 1: Experimental groups.

Group	Levels Active	Description
A	Level 1 only	Pure autoresearch, no outer intervention
B	Level 1 + 1.5	Inner loop plus outer strategy adjustment
C	Level 1 + 1.5 + 2	Full bilevel with mechanism research

Table 2: `val_bpb` change (Δ = best − baseline, more negative is better) over 30 inner iterations. Baseline `val_bpb` varies slightly across repeats due to training randomness (range 1.094–1.114). Group C's mean improvement is 5× that of Group A and 6.4× that of Group B (by absolute $|\Delta|$).

Group	R1	R2	R3	Mean ± Std
A (Level 1)	−0.009	−0.008	−0.011	-0.009 ± 0.001
B (Level 1 + 1.5)	−0.000	−0.010	−0.009	-0.007 ± 0.006
C (Level 1 + 1.5 + 2)	**−0.065**	−0.011	**−0.058**	$\mathbf{-0.045 \pm 0.030}$

3.7 Experimental Design

Three groups isolate the contribution of each level (table 1). All variables are held constant across groups: LLM model, GPU hardware (RTX 5090 32 GB, three independent servers), 300-second training budget, 30-iteration search budget, and baseline `train.py`. Each group runs 3 independent repeats; `train.py` is restored to the original baseline between repeats (verified by log inspection). The primary metric is Δ = best − baseline `val_bpb` (more negative indicates greater improvement).

4 Results

4.1 Primary Ablation Results

Table 2 reports `val_bpb` improvement for each group across three independent repeats.

Group A achieves consistent but small improvements: -0.009 ± 0.001. Group B is comparable to Group A (-0.007 ± 0.006); its R1 found essentially no improvement (−0.000), inflating variance. Group C achieves -0.045 ± 0.030, a 5× improvement over Group A. Two of three Group C repeats (R1 and R3) show dramatic gains (−0.065 and −0.058); R2 underperformed at −0.011.

4.2 Level 2 Mechanism Inventory

Table 3 lists all mechanisms generated by Level 2 across the six research sessions (two per repeat in Group C).

The three active named mechanisms are: **Tabu Search Manager** (maintains a tabu list of recently visited parameter regions, preventing the LLM from reproposing the same changes); **Multi-Scale Bandit Proposer** (treats parameter selection as a multi-armed bandit, balancing exploration and exploitation across parameters at different scales); and **Systematic Orthogonal Exploration** (forces the LLM to explore orthogonal parameter dimensions, preventing over-focus on a single parameter). Each mechanism was drawn from a different domain; Level 2 was not told which domains to consider.

Table 3: Level 2 mechanism inventory. All code was generated on the first attempt (zero retries). Five of six mechanisms passed import validation and were activated; one (GP Regressor) was reverted due to a missing `sklearn` dependency.

Repeat	Round	Mechanism	Domain	Import	Active
R1	1	Tabu Search Manager	Combinatorial opt.	✓	✓
R1	2	Helper class	—	✓	✓
R2	1	Multi-Scale Bandit Proposer	Online learning / MAB	✓	✓
R2	2	Helper class	—	✓	✓
R3	1	GP Regressor	Bayesian optimization	✓*	reverted
R3	2	Systematic Orthogonal Exploration	DOE	✓	✓

* Code valid but `sklearn` not installed; reverted automatically.

4.3 Search Behavior Analysis

The three groups exhibit qualitatively different search trajectories.

Group A: near-deterministic repetition. All three repeats follow nearly the same proposal sequence from the same baseline: iteration 1 attempts `TOTAL_BATCH_SIZE` increase (discard); iteration 2 reduces `WEIGHT_DECAY` (keep, $\Delta \approx -0.008$); iteration 3 sets `WINDOW_PATTERN="SSSS"` (keep, $\Delta \approx -0.002$); iterations 4–30 repeat these same two changes, accumulating up to 22 consecutive discards. The LLM, given the same state, generates nearly the same proposals every time.

Group B: redirected but bounded. Level 1.5 correctly identifies stalled parameters and redirects search: by cycle 2–3 the outer loop freezes `WEIGHT_DECAY` and `WINDOW_PATTERN` and redirects toward `LR`, `UNEMBEDDING_LR`, `MATRIX_LR`, and `FINAL_LR_FRAC`. Group B explores more parameters than Group A, but achieves similarly sized improvements because it operates within the same structural keep/discard framework.

Group C: Level 2 mechanisms unlock new directions. The decisive event in Group C's R1 and R3 is the discovery of `TOTAL_BATCH_SIZE` *reduction* (from 2^{19} to 2^{17} or 2^{18}), which produces improvements of -0.039 to -0.065—roughly 5–8$\times$ larger than any single change found by Groups A or B.

4.4 The `TOTAL_BATCH_SIZE` Discovery

The most impactful finding across all experiments is that reducing `TOTAL_BATCH_SIZE` from 2^{19} to 2^{17}–2^{18} dramatically improves `val_bpb` on the RTX 5090 under a 300-second training budget. The mechanism is straightforward: a smaller batch size yields more gradient steps within the fixed time budget, and better convergence for this 50M-parameter model. The original 2^{19} batch size was tuned for H100 throughput; the RTX 5090 running SDPA (Flash Attention 3 is unsupported on Blackwell compute 12.0) has different optimal batch characteristics.

Groups A and B both miss this direction for the same reason: DeepSeek's default search path attempts `TOTAL_BATCH_SIZE` *increase* first (an implicit "larger batch is better" bias). After the increase is discarded, Group A repeats it; Group B's outer loop freezes `TOTAL_BATCH_SIZE` after the failed increase, blocking the decrease direction entirely. Only Group C's Level 2 mechanisms—specifically, Tabu Search (which prevents revisiting failed directions) and Orthogonal Exploration (which forces dimensional diversity)—pushed the LLM to try the decrease direction.

5 Discussion

5.1 Hypothesis Testing

The experimental design is motivated by three hypotheses.

H1 (Group B > Group A): Not supported. Group B's mean improvement (-0.007 ± 0.006) is numerically worse than Group A's (-0.009 ± 0.001), though the difference is not meaningful given $n = 3$. The outer loop (Level 1.5) increases search diversity—Group B explores more parameters than Group A—but this diversity does not translate into larger improvements within the 30-iteration budget. Group B's R1 achieved essentially zero improvement (-0.000), the worst outcome of any repeat in any group. The outer loop correctly froze stalled parameters but, having done so, the LLM found nothing better in the remaining search space.

H2 (Group C > Group B): Supported. Group C's mean absolute improvement (-0.045 ± 0.030) is 6.4× Group B's (-0.007 ± 0.006). Despite high variance (± 0.030), two of three repeats produced dramatic improvements (-0.065, -0.058), and the separation between Groups C and B is large relative to the within-group variance, providing meaningful evidence that Level 2 adds value beyond Level 1.5.

H3 (Level 2 discovers novel mechanisms autonomously): Supported. Across three independent repeats, Level 2 generated mechanisms from four distinct domains (combinatorial optimization, online learning, Bayesian optimization, DOE) without being told which domains to consider. Code generation succeeded on the first attempt in all six sessions (zero retries). Five of six mechanisms passed import validation and were activated.

5.2 Why Level 1.5 Alone Is Insufficient

Level 1.5 can diagnose search problems—it correctly identifies when a parameter has been attempted repeatedly without improvement—and it can redirect search by adjusting which parameters are active. However, it is structurally limited to operating within the existing search framework. It cannot change how proposals are generated, modify the acceptance criterion, add pre-filtering steps, or introduce new search paradigms such as tabu lists, bandit selection, or orthogonal sampling. Only Level 2 can produce such code changes. This structural ceiling explains why H1 is not supported: parameter redirection alone is insufficient to escape the LLM's deterministic proposal patterns.

5.3 Why Group C's R2 Underperformed

Group C's R2 achieved only -0.011 improvement, comparable to Groups A and B, despite receiving Level 2 mechanisms. The most likely explanation is mechanism quality: R2's Level 2 generated the Multi-Scale Bandit Proposer, which—while valid and correctly injected—may be less effective than R1's Tabu Search Manager or R3's Orthogonal Exploration at forcing exploration of the batch size dimension. A secondary factor is overhead: each Level 2 research session requires approximately 3 minutes of wall time (four LLM calls), reducing effective inner iterations. With two sessions per repeat, Group C has roughly 6 minutes less search time than Groups A and B, a minor but non-zero cost.

5.4 Limitations

Small sample size. Three repeats per group is insufficient for rigorous statistical comparison. Group C's standard deviation (± 0.030) is 67% of its mean, indicating high variability. Reliable estimates would require $n \geq 10$ repeats per group.

Baseline variance. Baseline `val_bpb` varies across repeats (1.094–1.114) due to training randomness from data ordering and weight initialization. Using $\Delta = \text{baseline} - \text{best}$ mitigates this but does not eliminate it; a lower baseline gives less headroom for improvement. Future work should use fixed random seeds or report baseline-normalized metrics.

Single benchmark. All results are on one task: GPT pretraining at 50M parameters with a 300-second budget on RTX 5090. Generalization to other model sizes, training budgets, or tasks is unproven.

Dynamic load fragility. A preliminary run was invalidated because the Level 2 dynamic loading pipeline contained a `sys.modules` registration bug, causing all mechanism injections to silently fall back to the original runner. We fixed the bug before conducting the three reported repeats. This episode highlights the fragility of runtime code injection: silent fallback without error is a dangerous failure mode.

External dependency exposure. Level 2 has no constraint preventing it from importing external libraries. One of six generated mechanisms (GP Regressor) required `sklearn`, which was not installed. The validate-and-revert mechanism handled this correctly, but the exposure to arbitrary dependencies remains a reliability risk.

5.5 Unexpected Findings

Group A's search is near-deterministic. Given the same baseline `train.py`, DeepSeek generates almost the same proposal sequence across all three repeats. Group A's three repeats are therefore not independent samples in the usual sense: they are three runs of the same near-deterministic process, perturbed only by training noise. This determinism is itself a finding: LLMs hold strong priors about which hyperparameter changes to try first, and the inner loop alone cannot escape those priors.

Level 2 selects diverse domains without guidance. The Level 2 prompt instructs the model to "propose mechanism improvements" but does not suggest which algorithmic domains to consider. Across three independent repeats, the model independently chose combinatorial optimization, online learning, Bayesian optimization, and design of experiments. The LLM's breadth of knowledge therefore enables genuine exploration at the mechanism level, not just at the configuration level.

5.6 Future Work

Key directions for extending this work include: (1) scaling to $n \geq 10$ repeats with fixed random seeds for statistical power; (2) constraining Level 2 to pure-Python code without external imports, avoiding dependency failures; (3) evaluating on multiple benchmarks (different model sizes, tasks, budgets) to assess generalization; (4) adding a fourth group (Level 1 with `simple_mode=False`) to isolate the contribution of pre-baked mechanisms vs. autonomously discovered ones; and (5) investigating whether Level 2's code generation quality can be improved by providing a richer interface specification or a test harness.

6 Conclusion

We presented Bilevel Autoresearch, a bilevel framework in which an outer loop autonomously discovers and injects structural changes into an inner autoresearch loop. The key result is that Level 2—an autonomous mechanism research agent that writes and loads Python code at runtime—produces a 5× improvement in `val_bpb` over the Level-1 baseline on Karpathy's GPT pretraining benchmark, using the same DeepSeek model at every level. Level 1.5 alone does not reliably improve over the pure inner loop, confirming that parameter-level adjustment is insufficient to escape the LLM's deterministic search patterns; structural modification of the loop is required.

Level 2's mechanisms—Tabu Search Manager, Multi-Scale Bandit Proposer, and Systematic Orthogonal Exploration—were drawn from four distinct algorithmic domains and discovered without human specification, across three independent repeats. These mechanisms succeeded by forcing exploration of a search direction (`TOTAL_BATCH_SIZE` reduction) that the LLM's default priors systematically avoid.

The core principle is validated: *autoresearch can research itself.* The outer loop need not be human-designed; the same model that runs the inner loop can generate it, discovering structural improvements that previously required a human researcher to write.

References

[1] AIMing Lab. AutoResearchClaw: Multi-batch parallel autoresearch. https://github.com/aiming-lab/AutoResearchClaw, 2026. GitHub repository.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[3] Benoît Colson, Patrice Marcotte, and Gilles Savard. An overview of bilevel optimization. *Annals of Operations Research*, 153(1):235–256, 2007. doi: 10.1007/s10479-007-0176-2.

[4] EvoScientist Contributors. EvoScientist: Autoresearch with persistent experience memory. https://github.com/EvoScientist/EvoScientist, 2026. GitHub repository.

[5] Matthias Feurer and Frank Hutter. Hyperparameter optimization. In *Automated Machine Learning: Methods, Systems, Challenges*, pages 3–33. Springer, 2019.

[6] Luca Franceschi, Paolo Frasconi, Saverio Salzo, Riccardo Grazzi, and Massimiliano Pontil. Bilevel programming for hyperparameter optimization and meta-learning. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 1563–1572, 2018.

[7] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. Meta-learning in neural networks: a survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(9):5149–5169, 2021. doi: 10.1109/TPAMI.2021.3079209.

[8] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization (LION)*, pages 507–523, 2011. doi: 10.1007/978-3-642-25566-3_40.

[9] Andrej Karpathy. autoresearch: LLM-guided hyperparameter search for GPT pretraining. https://github.com/karpathy/autoresearch, 2026. GitHub repository.

[10] Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O. Stanley. Evolution through large models. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO)*, 2023.

[11] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158.

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

[13] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. In *International Conference on Learning Representations (ICLR)*, 2019.

[14] John R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976. doi: 10.1016/S0065-2458(08)60520-3.

[15] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M.P̃awan Kumar, Emilien Dupont, Francisco J.R̃. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024. doi: 10.1038/s41586-023-06924-6.

[16] Ankur Sinha, Pekka Malo, and Kalyanmoy Deb. A review on bilevel optimization: from classical to evolutionary approaches and applications. *IEEE Transactions on Evolutionary Computation*, 22(2):276–295, 2018. doi: 10.1109/TEVC.2017.2712906.

[17] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.