# Automated Prompt Optimization (APO) and Agentic Architecture: A Comprehensive Research Report

## 1. Executive Summary

The widespread adoption of Large Language Models (LLMs) has precipitated a fundamental shift in how artificial intelligence systems are engineered. The industry is rapidly transitioning from the manual, heuristic-based practice of "prompt engineering" to a more rigorous, algorithmic discipline known as **Automated Prompt Optimization (APO)**. This evolution mirrors the historical progression of software development from assembly language—where programmers manually managed memory and registers—to high-level compilers that automatically optimize code for performance and efficiency. In the context of Generative AI, APO represents the development of "compilers" for natural language, where the focus shifts from crafting specific instruction strings to defining intent, constraints, and evaluation metrics.

This report provides an exhaustive analysis of the current state of APO, synthesizing research on seminal tools including **APE (Automatic Prompt Engineer)**, **OPRO (Optimization by PROmpting)**, **PromptBreeder**, **DSPy**, and **TextGrad**. It distinguishes between three critical layers of interaction: **Autoprompting** (optimizing task instructions), **Meta-Autoprompting** (optimizing the optimization strategy itself), and **Recursive Autoprompting** (iterative self-improvement loops). By rigorously examining the mechanics of gradient-based text optimization, evolutionary strategies, and declarative compilation, this document establishes a production-grade framework for building self-improving agents capable of adapting to complex, dynamic environments without manual intervention.

The analysis culminates in a **Unified APO Framework**, a three-layer architectural blueprint—comprising the Task Harness, Prompt Optimizer, and Meta-Optimizer—that provides a concrete roadmap for engineering teams. This framework integrates advanced concepts such as "textual gradients" and "self-referential evolution" into a cohesive system, addressing critical implementation challenges such as prompt drift, reward hacking, and the computational costs of recursive optimization.

## 2. Theoretical Foundations of Automated Prompt Optimization

To understand the trajectory of APO, one must first deconstruct the theoretical underpinnings that enable LLMs to function as optimizers. The literature suggests that the optimization of natural language prompts can be framed analogously to mathematical optimization in continuous spaces, yet it operates within the discrete and semantic, non-differentiable space of language tokens.

### 2.1 The Prompt Optimization Problem

Formally, the prompt optimization problem can be defined as finding an optimal instruction  that maximizes a scoring function  over a dataset of inputs  and targets , where  is the LLM. Unlike traditional machine learning, where parameters (weights) are updated via gradient descent, APO treats the *prompt itself* as the trainable parameter.1

This presents unique challenges. The search space of natural language is combinatorial and vast (, where  is vocabulary size and  is sequence length). Furthermore, the optimization landscape is non-convex and non-differentiable; a single token change can drastically alter the semantic meaning and downstream performance of the model. APO methodologies attempt to navigate this landscape using various strategies, which broadly cluster into **Search-Based**, **Gradient-Based**, and **Evolutionary** approaches.

### 2.2 Taxonomy of Optimization Strategies

The research identifies distinct families of APO algorithms, each leveraging different mechanisms to traverse the prompt search space.

| **Strategy Family** | **Core Mechanism** | **Representative Tools** | **Search Landscape** |
| --- | --- | --- | --- |
| **Instruction Induction (Search)** | Generates candidate instructions from input-output pairs using an LLM, then selects the best via evaluation. | **APE** 2 | Discrete; Candidate-based selection. |
| **Trajectory-Based Optimization** | Uses the history of past attempts (trajectory) in the prompt to guide the LLM toward better solutions. | **OPRO** 3 | Iterative; Path-dependent exploration. |
| **Evolutionary Algorithms** | Applies genetic operators (mutation, crossover) to a population of prompts to evolve fitness over generations. | **PromptBreeder**, **EvoPrompt** 4 | Population-based; Global search. |
| **Textual Gradients (Differentiation)** | Backpropagates textual critiques (simulated gradients) from output to input to update components. | **TextGrad** 1 | Gradient-descent analogue; Directed local search. |
| **Declarative Compilation** | Optimizes modular programs by bootstrapping demonstrations and refining instructions against a metric. | **DSPy** 5 | Bayesian Optimization / Bootstrapping. |
| **System Prompt Engineering** | Optimizes the global instruction layer using component libraries and heuristic search. | **SPRIG** 6 | Combinatorial; Component-based assembly. |

### 2.3 The Shift to Meta-Optimization

A critical theoretical advancement is the move from simple optimization (improving ) to **Meta-Optimization** (improving the algorithm that improves ). Research into tools like **PromptBreeder** and **metaTextGrad** demonstrates that fixed optimization strategies (e.g., "paraphrase this prompt") often plateau. By optimizing the *mutation operators* or the *optimization structure* itself, systems can break out of local optima and adapt to novel domains.7 This capability is essential for "General Purpose" agents that must operate across diverse tasks without manual retuning.

## 3. Deep Analysis of Primary APO Methodologies

This section provides a granular analysis of the mechanics, strengths, and failure modes of the primary APO tools identified in the research.

### 3.1 APE: Automatic Prompt Engineer

**Automatic Prompt Engineer (APE)** represents the foundational "Generation + Selection" approach to APO. It treats instruction generation as a program synthesis problem, leveraging the LLM's inherent ability to infer intent from demonstrations.

#### 3.1.1 Algorithmic Mechanics

APE operates in a two-stage process:

1. **Candidate Generation (Proposal):** The system prompts an LLM (the inference model) with a set of input-output pairs (demonstrations) and a meta-template such as: *"I gave a friend an instruction and five inputs. The friend read the instruction and wrote an output for every one of the inputs. Here are the input-output pairs... The instruction was ."*.2 This effectively asks the model to reverse-engineer the latent instruction that explains the data.
2. **Score-Based Selection:** The generated candidates are then evaluated. APE executes each candidate instruction on a validation set. The selection criterion typically uses a score function, such as execution accuracy (for deterministic tasks) or log-probability (for open-ended generation), to rank the candidates.

#### 3.1.2 Key Insights and Performance

Research indicates that APE is capable of discovering zero-shot prompts that outperform human-engineered baselines. A notable finding from the APE paper was the discovery of the prompt *"Let's work this out in a step by step way to be sure we have the right answer,"* which achieved superior performance on reasoning tasks compared to the standard "Let's think step by step".2 This demonstrates that subtle semantic variations—such as adding "to be sure we have the right answer"—can trigger more robust reasoning pathways in the model.

#### 3.1.3 Limitations for Agentic Systems

While effective for bootstrapping, APE has limitations in dynamic agentic contexts. It primarily optimizes for a static task description based on a fixed dataset. It lacks a mechanism for continuous, online adaptation once the prompt is selected. Furthermore, its performance relies heavily on the quality and representativeness of the few-shot examples provided during the induction phase. If the examples are biased, the induced prompt will be brittle.

### 3.2 OPRO: Optimization by PROmpting

**Optimization by PROmpting (OPRO)** advances the field by utilizing the LLM not just as a generator, but as the *optimizer algorithm* itself. It introduces the concept of **history-in-prompt** optimization.

#### 3.2.1 Algorithmic Mechanics

OPRO abandons formal mathematical solvers in favor of natural language iteration. The core mechanism involves a "meta-prompt" that contains the **optimization trajectory**.

* **The Meta-Prompt:** This prompt includes a description of the optimization problem and a historical log of previously generated instructions paired with their optimization scores (e.g., accuracy on a training set).
* **The Optimization Step:** The LLM reads this trajectory, identifies patterns in high-scoring vs. low-scoring instructions, and generates new candidate instructions intended to achieve a higher score.
* **The Loop:** These new candidates are evaluated, scored, and added to the trajectory, closing the loop.

#### 3.2.2 The "Take a Deep Breath" Phenomenon

OPRO's ability to traverse the optimization landscape led to the discovery of counter-intuitive prompts. For example, on the GSM8K math benchmark, OPRO discovered that appending *"Take a deep breath"* to the prompt significantly improved performance.2 This highlights the opacity of the LLM's internal optimization surface; the model responds to psychological priming in ways that human engineers might not predict.

#### 3.2.3 Limitations and Model Scale Sensitivity

A critical nuance in OPRO's deployment is its sensitivity to the capability of the optimizer LLM. Research revisiting OPRO with smaller models (e.g., Llama-2-7B, Mistral-7B) reveals a sharp performance degradation.9 Smaller models often struggle to interpret the long-context optimization trajectory effectively, failing to deduce the gradient of improvement. They may hallucinate scores or repeat previous failures. Thus, OPRO is best implemented using frontier-class models (e.g., GPT-4, Claude 3.5) as the optimizer, even if the target model (the one being optimized) is smaller.

### 3.3 PromptBreeder: Self-Referential Evolutionary Systems

**PromptBreeder** represents a leap toward **Meta-Autoprompting**. Unlike APE or OPRO, which optimize a static variable (the prompt), PromptBreeder optimizes the *method of variation* itself.

#### 3.3.1 Algorithmic Mechanics: The Evolutionary Loop

PromptBreeder employs a genetic algorithm that manages a population of "evolutionary units." Each unit contains not just a **Task-Prompt** () but also a **Mutation-Prompt** ()—the instruction used to modify .

The system evolves through a binary tournament selection process:

1. **Initialization:** A population is seeded with task descriptions, general "thinking styles" (cognitive heuristics like "Think step-by-step"), and basic mutation prompts.
2. **Fitness Evaluation:** Units are evaluated on a batch of training data.
3. **Selection & Mutation:** Winners of the tournament are replicated and mutated.

#### 3.3.2 The Five Classes of Mutation Operators

PromptBreeder's distinct innovation lies in its diverse set of mutation operators, which allow for both gradual refinement and radical structural changes 7:

1. **Direct Mutation:**
   * *Zero-order:* Generates a new prompt from scratch using the problem description.
   * *First-order:* Applies the mutation-prompt  to the task-prompt  (e.g., : "Make this more concise"  : "Solve X"  : "Solve X briefly").
2. **Estimation of Distribution (EDA):** The LLM is shown a list of high-performing prompts from the population history and asked to generate a new prompt that follows the distribution of success. This effectively "interpolates" between successful strategies.
3. **Hypermutation (Meta-Optimization):** This operator mutates the *Mutation-Prompt* itself. For example, it might evolve the mutator from *"Paraphrase this"* to *"Rewrite this to be more formal and precise."* This is the self-referential engine that allows the system to improve its own learning capability.
4. **Lamarckian Mutation:** This operator mimics Lamarckian evolution by reverse-engineering a prompt from a successful phenotype. The system takes a correct reasoning trace (the "working out") and asks the LLM, *"What instruction would have led to this solution?"* This allows the system to codify successful behaviors into instructions.
5. **Crossover and Shuffling:** Combines traits from two different parent prompts or reshuffles the context of few-shot examples to maintain population diversity.

#### 3.3.3 Agentic Utility

PromptBreeder is particularly valuable for **domain adaptation**. When an agent enters a novel domain where standard prompt engineering heuristics fail, PromptBreeder can evolve specialized "thinking styles" unique to that domain. However, the computational cost is significant due to the large population size and multiple generations required for convergence.4

### 3.4 DSPy: Declarative Self-improving Python

**DSPy** fundamentally reframes the interaction paradigm. It moves away from "string manipulation" toward "programming with signatures." It is less an optimization algorithm and more a **compiler infrastructure** for LLM pipelines.

#### 3.4.1 Mechanics: Signatures, Modules, and Teleprompters

DSPy introduces three core abstractions 5:

* **Signatures:** Declarative specifications of input/output behavior (e.g., question -> answer, context, question -> rational, answer). These replace free-form prompt strings.
* **Modules:** Python classes (e.g., dspy.ChainOfThought, dspy.ReAct) that implement the logic of how the signature is executed.
* **Teleprompters (Optimizers):** Algorithms that "compile" the modules. The compiler optimizes the prompt instructions and, crucially, selects effective **few-shot demonstrations** (bootstrapping) to maximize a metric.

#### 3.4.2 The MIPROv2 Optimizer

The **MIPROv2 (Multi-prompt Instruction Proposal Optimizer)** is a state-of-the-art teleprompter within DSPy. It employs a Bayesian Optimization approach to jointly optimize instructions and few-shot examples.11

1. **Bootstrapping:** It runs the program on a training set to collect "traces" (execution logs). It filters these traces to find successful examples (where the metric was satisfied).
2. **Proposal:** It generates a set of candidate instructions and candidate sets of few-shot examples derived from the successful traces.
3. **Search:** It uses a Bayesian surrogate model to explore the combinatorial space of (Instruction, Example\_Set) pairs, efficiently finding the combination that maximizes the metric.

#### 3.4.3 Integration into Agents

DSPy is the ideal **runtime architecture** for agentic systems. It allows developers to define the agent's logic (the graph of modules) independently of the prompt optimization. An agent built with DSPy can be "recompiled" for different models (e.g., moving from GPT-4 to Llama-3) simply by re-running the teleprompter, ensuring the prompts are always optimized for the specific weights of the backend model.

### 3.5 TextGrad: Automatic Differentiation via Text

**TextGrad** introduces the paradigm of **Textual Gradients**, adapting the backpropagation algorithm of deep learning to the semantic space of text.1

#### 3.5.1 Mechanics: The Computation Graph

TextGrad models the agentic system as a computation graph where nodes are variables (prompts, inputs, outputs) and edges are operations (LLM calls, tool usage).

* **Forward Pass:** The system executes the graph to produce an output.
* **Loss Computation:** A "TextLoss" function evaluates the output (e.g., via an LLM judge or programmatic check) and generates a critique.
* **Backward Pass (Textual Backpropagation):** The critique is propagated backward through the graph. At each node, the LLM acts as a "gradient operator," determining how the variable (e.g., the system prompt) contributed to the error and generating a specific "textual gradient" (feedback) to correct it.
* **Update Step:** The optimizer applies the textual gradient to modify the prompt, similar to how SGD updates weights.

#### 3.5.2 metaTextGrad: Optimizing the Optimizer

A significant extension, **metaTextGrad**, applies this logic recursively. It defines a **Meta-Structure Optimizer** that optimizes the architecture of the optimization pipeline itself.8

* **Inner Loop:** Optimizes the specific program (agent) using textual gradients.
* **Outer Loop:** Optimizes the *optimizer*. It refines the prompts used to generate critiques and explores different structures for combining optimizers. This hierarchical approach allows the system to learn *how* to critique effectively for a specific domain.

#### 3.5.3 Utility for Multi-Agent Systems

TextGrad is uniquely suited for **debugging complex multi-agent workflows**. In a chain of agents (e.g., Planner -> Coder -> Reviewer), if the final code fails, it is often unclear which agent is at fault. TextGrad's backpropagation can identify that the *Planner* provided a vague spec, generating a gradient specifically for the Planner's prompt, rather than just blaming the Coder.

### 3.6 SPRIG: System Prompt Optimization

**SPRIG (System PRompt Improvement through Genetic algorithm)** specifically targets the optimization of the **System Prompt**—the global instruction that defines the agent's persona and constraints.6

#### 3.6.1 Mechanics: Component-Based Evolution

SPRIG utilizes a curated corpus of semi-structured prompt components (e.g., Roles, Tones, Safety Constraints, Thinking Styles).

* **Genetic Algorithm:** It employs an edit-based genetic algorithm with operations like **Add**, **Swap**, **Delete**, and **Rephrase** to modify the system prompt using components from the corpus.
* **UCB Pruning:** To manage the search space, it uses the **Upper Confidence Bound (UCB)** algorithm to select the most promising components for exploration, balancing exploration of new components with exploitation of known good ones.

#### 3.6.2 Comparison with ProTeGi

Research comparing SPRIG to **ProTeGi** (Prompt Optimization with Textual Gradients) highlights a key distinction: ProTeGi samples multiple "gradients" to generate candidates, which works well for task-specific prompts. However, SPRIG's component-based approach produces system prompts that demonstrate superior **generalization**.15 A SPRIG-optimized system prompt improves performance across a wide range of unseen tasks, whereas ProTeGi-optimized prompts often overfit to the specific training task. This makes SPRIG essential for defining the "constitution" of a general-purpose agent.

## 4. The Unified Framework for Automated Prompt Optimization

To operationalize these diverse methodologies, this report defines a **Unified APO Framework**. This framework organizes the components into a three-layer reference architecture that can be implemented in production systems.

### 4.1 Layer A: Task Harness (The Source of Truth)

This layer provides the grounding for all optimization. It defines the "loss function" of the agentic system.

* **Function:** Serves as the immutable standard against which candidates are measured.
* **Components:**
  + **Datasets:** Labeled examples (input/output) for training and validation.
  + **Metric Suite:** A combination of **Programmatic Metrics** (e.g., unit tests, code compilation, JSON schema validation) and **LLM-Based Judges** (e.g., "Rate the helpfulness on a scale of 1-5").
  + **Hard Constraints:** Binary pass/fail checks for safety, latency, and token budget.
  + **Logging:** A structured ledger (e.g., SQL/NoSQL) recording every (prompt, input, output, score) tuple.
* **Integration:** This layer mirrors the evaluation loop in APE and the metric definitions in DSPy.

### 4.2 Layer B: Prompt Optimizer (The Search Engine)

This layer executes the active search loop. It treats the prompt as a variable to be optimized against Layer A.

* **Function:** Iteratively improves the prompt using a specific algorithm.
* **Components:**
  + **Candidate Generator:** The engine that produces new prompt variations. This can be an implementation of **OPRO** (using trajectory history), **APE** (using induction), or **TextGrad** (using textual backpropagation).
  + **Evaluator:** The interface that sends candidates to Layer A and retrieves scores.
  + **Selector:** The logic for determining which candidates survive (e.g., Top-K selection, Tournament selection).
  + **History/Memory:** A buffer storing the "optimization trajectory" required by algorithms like OPRO.

### 4.3 Layer C: Meta-Optimizer (The Controller)

This layer provides adaptive intelligence, optimizing the configuration of Layer B.

* **Function:** "Learns how to learn." It updates the strategy of optimization based on meta-performance.
* **Components:**
  + **Mutation Prompt Editor:** Updates the instructions used by the Candidate Generator (e.g., switching from "Paraphrase" to "Think step-by-step"). This implements **PromptBreeder's Hypermutation**.
  + **Rubric Editor:** Refines the instructions given to the LLM Judges in Layer A to better align with ground truth or human preference.
  + **Hyperparameter Scheduler:** Dynamically adjusts parameters like the LLM temperature (exploration rate), the number of candidates per round, or the token budget based on the rate of convergence.
  + **Structure Optimizer:** Dynamically reconfigures the agent pipeline (e.g., adding a "Reviewer" node) if performance plateaus, implementing **metaTextGrad's** structure optimization.

## 5. Agentic Interaction Patterns: Meta-Prompting and Reflexion

While APO optimizes the *components* of an agent, specific interaction patterns optimize the *process* of agency.

### 5.1 Meta-Prompting (Microsoft/Stanford)

**Meta-Prompting** is a specific orchestration pattern that transforms a single LLM into a "Conductor" and a set of "Experts".16

* **The Conductor:** A central persona responsible for decomposing complex tasks into sub-tasks and assigning them to experts.
* **The Experts:** Specialized instances of the same model (e.g., "Python Expert," "History Expert") invoked with specific instructions.
* **Fresh Eyes:** A critical innovation in Meta-Prompting is the "Fresh Eyes" principle. Unlike multi-persona prompting where all agents share a context window, experts in Meta-Prompting are often invoked with isolated contexts. This prevents "context pollution" where an expert double-downs on errors made by previous agents. The Conductor integrates their isolated outputs.
* **Integration with APO:** The Unified Framework can optimize this structure. **SPRIG** can be used to optimize the Conductor's decomposition instructions. **DSPy** can be used to compile optimal instructions for each Expert role.

### 5.2 Reflexion vs. Self-Refine

These patterns represent **Recursive Autoprompting** at the inference level.

* **Self-Refine:** An iterative loop where the model generates an output, critiques it, and refines it. It is typically intra-episode; the refinement happens within a single task execution.18
* **Reflexion:** Adds a persistence layer. It converts feedback into a verbal "self-reflection" (e.g., "I failed because I used the wrong tool. Next time I should use the Search tool"). This reflection is stored in an **Episodic Memory** and retrieved in *future* trials.19
* **Distinction:** Self-Refine optimizes the *current* output. Reflexion optimizes the *agent's policy* (via memory) for future tasks. Reflexion acts as a form of "verbal reinforcement learning."

## 6. Implementation Guide & Engineering Constraints

Implementing the Unified APO Framework requires addressing significant engineering challenges regarding cost, latency, and stability.

### 6.1 Data Schema and Logging

A robust implementation requires a standardized schema to track the optimization process.

* **RunState:** Stores the current iteration, budget used, and active candidates.
* **PromptCandidate:** id, parent\_id, text, generation\_method (e.g., "mutation\_operator\_3"), score, trace\_log.
* **MetaState:** Stores the current configuration of the optimizer (e.g., current mutation prompts, current temperature).

### 6.2 Latency and Cost Management

Recursive optimization loops can be prohibitively expensive.

* **Asynchronous Optimization:** The "Optimization Layer" (Layer B) should operate asynchronously from the runtime agent. The agent executes using cached, frozen prompts. The optimizer runs in the background (e.g., overnight), consuming logs of the day's interactions to generate improved prompts for the next deployment.
* **Budget Caps:** Strict stop conditions must be implemented.
  + *Plateau Detection:* Stop if the moving average of the score improves by less than  (e.g., 0.1%) over  rounds.
  + *Token Cap:* Hard stop if accumulated token usage exceeds a defined budget.

### 6.3 Drift Detection: Semantic Constraint Regression

**Prompt Drift** occurs when an optimized prompt achieves a higher metric score but loses the original semantic intent or violates safety constraints (e.g., a prompt becoming overly verbose to satisfy a "detail" metric, ignoring a "conciseness" constraint).11

* **Algorithm:** **Semantic Constraint Regression (SCR)**
  1. **Baseline Embedding:** Compute the vector embedding of the original, human-verified prompt ().
  2. **Similarity Check:** For every candidate , calculate Cosine Similarity .
  3. **Constraint Unit Tests:** Execute a suite of specific assertions on the prompt text itself (e.g., assert "JSON" in prompt, assert "polite" in prompt).
  4. **Trigger:** If  or if any constraint test fails, the candidate is flagged as "Drifted" and discarded, regardless of its performance score.

## 7. Risk Register and Governance

Automating the prompt engineering loop introduces novel risks.

| **Failure Mode** | **Description** | **Mitigation Strategy** |
| --- | --- | --- |
| **Reward Hacking** | The optimizer finds a "cheat code" that satisfies the metric without solving the task (e.g., returning empty but valid JSON). | Use **Rubric Diversity** (multiple judges) and **Hard Constraints** (unit tests) alongside LLM judges. |
| **Evaluator Leakage** | The prompt optimizer learns to copy the evaluation rubric into the task prompt, confusing the agent. | Sanitize candidates; strictly separate Task Context from Evaluation Context. |
| **Recursion Runaway** | In Reflexion loops, the agent gets stuck in a cycle of "I made a mistake, I will fix it" without fixing it. | Limit recursion depth (max\_retries). Implement **Fresh Eyes** meta-prompting to reset context. |
| **Prompt Injection** | If the optimizer consumes user data to generate prompts, it is vulnerable to injection attacks. | Never use untrusted user data in the optimization set. Use only curated "Gold" datasets. |

## 8. Conclusion and Future Outlook

The field of Automated Prompt Optimization has matured from simple heuristic search to sophisticated, self-referential compiler architectures. The convergence of **DSPy** (as the runtime infrastructure), **TextGrad** (for granular, gradient-based debugging), and **PromptBreeder** (for open-ended evolutionary adaptation) suggests a future where human involvement in prompt engineering is reduced to defining the *intent* and the *metrics* of the system.

For immediate implementation, engineering teams are advised to adopt a **DSPy-based architecture** as the foundation. This facilitates the integration of **MIPROv2** for robust instruction tuning. For complex, multi-agent workflows, a **TextGrad** layer should be employed during the development phase to debug inter-agent communication. Finally, a **Meta-Cognitive Layer** utilizing **PromptBreeder-style** evolutionary logic should be established to ensure long-term adaptability and resistance to domain drift. The agent of the future will not merely be "prompted"; it will be "compiled" and "evolved."

**Sources:** 2 APE Mechanics & Zero-Shot CoT 3 OPRO Mechanics & Small Model Limits 4 PromptBreeder Mutation Operators 1 TextGrad & metaTextGrad Algorithms 5 DSPy, MIPROv2, & Compilation 6 SPRIG & System Prompt Optimization 16 Meta-Prompting & Conductor/Experts 18 Reflexion vs. Self-Refine 20 Unified Framework Definitions 11 Drift Detection & Constraints

#### Works cited

1. [PDF] TextGrad: Automatic "Differentiation" via Text - Semantic Scholar, accessed February 7, 2026, <https://www.semanticscholar.org/paper/TextGrad%3A-Automatic-%22Differentiation%22-via-Text-Yuksekgonul-Bianchi/f91c987921cbb63666d7f22a9071693e51e8d16a>
2. Automatic Prompt Engineer (APE), accessed February 7, 2026, <https://www.promptingguide.ai/techniques/ape>
3. Large Language Models as Optimizers - arXiv, accessed February 7, 2026, <https://arxiv.org/pdf/2309.03409>
4. Promptbreeder: Self-Referential Self-Improvement via Prompt Evolution | OpenReview, accessed February 7, 2026, <https://openreview.net/forum?id=HKkiX32Zw1>
5. DSPy: Compiling Declarative Language Model Calls into State-of-the-Art Pipelines, accessed February 7, 2026, <https://hai.stanford.edu/research/dspy-compiling-declarative-language-model-calls-into-state-of-the-art-pipelines>
6. [2410.14826] SPRIG: Improving Large Language Model Performance by System Prompt Optimization - arXiv, accessed February 7, 2026, <https://arxiv.org/abs/2410.14826>
7. [2309.16797] Promptbreeder: Self-Referential Self-Improvement Via Prompt Evolution, accessed February 7, 2026, <https://arxiv.org/abs/2309.16797>
8. metaTextGrad: Automatically optimizing language model ... - arXiv, accessed February 7, 2026, <https://arxiv.org/abs/2505.18524>
9. [2405.10276] Revisiting OPRO: The Limitations of Small-Scale LLMs as Optimizers - arXiv, accessed February 7, 2026, <https://arxiv.org/abs/2405.10276>
10. [PDF] DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines, accessed February 7, 2026, <https://www.semanticscholar.org/paper/DSPy%3A-Compiling-Declarative-Language-Model-Calls-Khattab-Singhvi/2069aaaa281eb13bcd9330fc4d43f24f6b436a53>
11. Optimizers - DSPy, accessed February 7, 2026, <https://dspy.ai/learn/optimization/optimizers/>
12. Prompt Optimization with DSPy: GEPA Explained with Python Examples - Medium, accessed February 7, 2026, <https://medium.com/@melikedulkadir/prompt-optimization-with-dspy-gepa-explained-with-python-examples-e85f4ea17a8d>
13. arXiv:2406.07496v1 [cs.CL] 11 Jun 2024, accessed February 7, 2026, <https://3dvar.com/Yuksekgonul2024TextGrad.pdf>
14. arxiv.org, accessed February 7, 2026, <https://arxiv.org/html/2505.18524v1>
15. Sprig: Improving Large Language Model Performance by System Prompt Optimization, accessed February 7, 2026, <https://arxiv.org/html/2410.14826v2>
16. Meta-Prompting: Enhancing Language Models with Task-Agnostic ..., accessed February 7, 2026, <https://arxiv.org/abs/2401.12954>
17. Researchers from Stanford and OpenAI Introduce 'Meta-Prompting': An Effective Scaffolding Technique Designed to Enhance the Functionality of Language Models in a Task-Agnostic Manner - MarkTechPost, accessed February 7, 2026, <https://www.marktechpost.com/2024/01/27/researchers-from-stanford-and-openai-introduce-meta-prompting-an-effective-scaffolding-technique-designed-to-enhance-the-functionality-of-language-models-in-a-task-agnostic-manner/>
18. #12: How Do Agents Learn from Their Own Mistakes? The Role of Reflection in AI, accessed February 7, 2026, <https://huggingface.co/blog/Kseniase/reflection>
19. Reflexion: Language Agents with Verbal Reinforcement Learning - OpenReview, accessed February 7, 2026, <https://openreview.net/pdf?id=vAElhFcKW6>
20. chatgpt-autoprompting-research-refinement-2026-02-07-1770445108466.pdf
21. Meta-Prompt Optimization for LLM-Based Sequential Decision Making - arXiv, accessed February 7, 2026, <https://arxiv.org/html/2502.00728v1>
22. Promptbreeder: Self-Referential Self-Improvement via Prompt Evolution - arXiv, accessed February 7, 2026, <https://arxiv.org/html/2309.16797>
23. System Prompt Optimization with Meta-Learning - arXiv, accessed February 7, 2026, <https://arxiv.org/html/2505.09666v1>