machine learning & knowledge extraction

MDPI

Article

An Overview of Large Language Models and a Novel, Large Language Model-Based Cognitive Architecture for Solving Open-Ended Problems

Hashmath Shaik, Gnaneswar Villuri and Alex Doboli *

Department of Electrical and Computer Engineering, Stony Brook University, Stony Brook, NY 11794-2350, USA; hashmath.shaik@stonybrook.edu (H.S.); gnaneswar.villuri@stonybrook.edu (G.V.)
* Correspondence: alex.doboli@stonybrook.edu

Abstract

Large Language Models (LLMs) offer new opportunities to devise automated implementation generation methods that can tackle problem solving beyond traditional methods, which usually require algorithmic specifications and use only static domain knowledge. LLMs can support devising new methods to support activities in tackling open-ended problems, like problem framing, exploring possible solving approaches, feature elaboration and combination, advanced implementation assessment, and handling unexpected situations. This paper presents a detailed overview of the current work on LLMs, including model prompting, retrieval-augmented generation (RAG), and reinforcement learning. It then proposes a novel, LLM-based Cognitive Architecture (CA) to generate programming code starting from verbal discussions in natural language, a particular kind of problem-solving activity. The CA uses four strategies, three top-down and one bottom-up, to elaborate, adaptively process, memorize, and learn. Experiments are devised to study the CA performance, e.g., convergence rate, semantic fidelity, and code correctness.

Keywords: open-ended problem solving; Large Language Models; implementation generation; prompting; retrieval-augmented generation; reinforcement learning

check for updates

Academic Editor: Jianlong Zhou

Received: 24 September 2025
Revised: 21 October 2025
Accepted: 25 October 2025
Published: 1 November 2025

Citation: Shaik, H.; Villuri, G.; Doboli, A. An Overview of Large Language Models and a Novel, Large Language Model-Based Cognitive Architecture for Solving Open-Ended Problems. *Mach. Learn. Knowl. Extr.* **2025**, *7*, 134. https://doi.org/10.3390/make7040134

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Problem solving is the process of creating a solution for a problem description [1–4]. The solution can be an *explanation* for a set of properties exhibited by a static or dynamic situation, e.g., a mathematical proof or an *implementation* (realization), which is the construction of a new materialization (e.g., design) that exhibits the required properties as a result of their operation (functioning, execution). This report focuses on the implementation (realization) of problem solving.

Creating an implementation can pertain to the three general-purpose problem-solving situations: well-defined problems, ill-defined problems, and open-ended problems [5–7]:

1. Well-defined problem solving for implementation construction describes situations in which an existing solution can be reused with some incremental changes to solve a new problem. For example, textbook algorithms are utilized to solve a new problem by selecting proper data structures and customizing the algorithm parameters, like the conditions of conditional statements and the iterations of loops. Using parameterized templates for circuit design [8–11] belongs to this category too.

2. Ill-defined problem solving for implementation construction represents cases in which the existing implementations cannot solve all requirements, i.e., they satisfy some but not others [12,13]. Changing the parameters of the implementation does not address the issue. Problem solving includes options, like producing a description of the implementation trade-offs by parameter sampling and selecting the best compromise, exploring implementation alternatives for specific fragments of the implementation so that better trade-offs result for the overall solution, and selecting a different approach (principle) for an implementation, including situations when a new implementation must be built, similar to open-ended solving for building a new implementation.
3. Open-ended problem solving for implementation generation requires devising new solutions with distinct characteristics and a significant departure from previous implementations. The understanding of this process is still limited [14,15]. Also, there are insufficient metrics to describe the degree to which the process is systematically progressing towards success, e.g., building a new implementation. Typical activities include problem framing and problem understanding, identifying and selecting the solving approach, divide and conquer (e.g., problem partitioning into sub-problems), implementation elaboration through trial and error, feature combination, adjustment, abstraction and insight gaining, implementation analysis to find pros and cons and the impact of features on the implementation operation, implementation modification, error correction, and handling unexpected situations.

Devising automated methods for solving open-ended problems has been a main research topic for a long time [16], but current approaches are still insufficient [15]. Traditional methods mainly tackle well-defined and ill-defined problems and rely on approaches such as high-level synthesis (HLS), evolutionary algorithms, agent-based methods, and Cognitive Architectures (CAs). Section 2 elaborates each of the four approaches. HLS has proven to be the most effective option, even though it is customized for a specific application domain, hence more rigid, and requires developing a description of the system to be designed. CAs are the opposite approach as they are flexible domain-wise and do not require descriptions that follow the rules of a specification language. However, they require more advanced knowledge representations and algorithms, which have been less studied compared to the other alternatives. Current advances in Large Language Models (LLMs) offer intriguing opportunities to base novel CAs on LLMs, so that LLM capabilities in natural language processing (NLP), knowledge retrieval and summarization, and reasoning can be utilized. LLM-based CAs have not yet been studied for open-ended problem solving.

This paper proposes a novel, LLM-based CA for solving open-ended problems, in particular, to generate programming code based on verbal discussions in small teams, like three or four participants. Optimizing teamwork is an important current research topic [16]. The paper also offers a detailed overview of LLM methods, including prompting, retrieval-augmented generation (RAG), and reinforcement learning. The proposed CA implements the activities of elaboration, adaptive processing, and memory and learning through four strategies that address top-down and bottom-up solving. The paper presents the prompting and RAG-based algorithms to implement the four strategies. RAG uses a knowledge graph to create a structural representation used to infer the most important features used by the bottom-up solving strategy. Experiments are used to study the CA performance in code generation from verbal discussions. Its convergence rate, semantic fidelity, and code correctness are compared to those of other LLM-based generation methods.

The paper has the following structure. Section 2 summarizes the traditional methods for automated implementation generation. Section 3 offers an extensive overview of LLMs, prompt engineering, retrieval-augmented generation, and reinforcement learning.

Section 4 discusses a novel LLM-based CA for open-ended problem solving. Section 5 presents experimental results for CA. Conclusions end the paper.

2. Overview of Traditional Automated Implementation Generation

Traditional approaches to automatically generate implementations can be grouped into four broad categories: (i) approaches based on high-level specifications, (ii) methods using evolutionary algorithms, (iii) agent-based methods, and (iv) cognitive architectures. The four categories are summarized next.

(i) Approaches based on high-level specifications: These approaches include traditional compiling methods to generate executable code [17] and high-level synthesis methods [18–21] and template-based synthesis [10,11] to create electronic circuits and systems. They use high-level specifications described using a programming language. Conceptually, specifications serve as parameterized descriptions of the target implementation architecture. Specifically, internal representations are built using a set of predefined rules (e.g., language grammar) applied to the specifications and then used to create an optimized hardware design by exploring different optimization possibilities. Prediction models or simulation tools are integrated to evaluate the performance of possible implementation alternatives.

These methods address the problem-solving activities in the following ways: The specification gives an unambiguous, complete description of the parameterized architecture. Thus, there is no problem framing step and problem understanding is fully addressed during specification creation. Divide and conquer is defined by the structuring of the specification. Also, there is no step of exploring possible implementation alternatives, as the specification explicitly describes the data processing steps, including the connections between the sequences of processing steps, i.e., using the processing outputs as inputs for the next processing steps. Hence, feature combination during elaboration only connects predefined operators which do not change their function based on the connections. From the point of view of cognitive psychology, these combinations are relation-based combinations but do not reflect feature-based combinations, in which features of a concept are transferred to another concept [22]. Hence, there are no unexpected situations, including emerging features. Implementation analysis uses performance models and simulation, even though the pros and cons of an implementation are rarely causally linked to the implementation fragments responsible for them. Hence, the insight gain is limited. Trial and error (possibly guided by priority functions), implementation modification, and adjustment are only at the level of optimizing the architecture parameters. There is no abstraction or summarization during the process. Error correction requires to modify the specification and then repeat the problem-solving process.

(ii) Methods using evolutionary algorithms: These methods create a dynamic process, in which large populations of solutions originate new populations through traditional operators, i.e., selection, crossover, and mutation [23]. Selection means propagating high-fitness individuals from the current to the next population, crossover combines features of a set of solutions to produce new solutions, and mutation randomly changes solution features.

These methods do not include problem framing and understanding. Identifying and selecting the implementation approach has been studied less, even though it is possible to maintain separate sub-populations, each for a different approach, and then giving higher priority to the sub-populations that include more high-quality implementations. There is no divide and conquer to separate a problem into sub-problems and no explicit error correction. Trial and error is mimicked through the mutation operator, even though mutation does not implement a systematic exploration process guided by the learned knowledge. There is no insight gained during the process, abstraction or summarization of the learned knowledge, and no explicit identification of unexpected situations. Crossover

implements combinations, including feature and relation combinations. Similar to the previous category, implementation analysis uses performance models and simulation to produce a fitness value that controls the selection of the better implementations. However, there is no explicit identification of the causal features that produce the pros and cons of an implementation, thus there is no implementation adjustment, modification, or correction guided by causal information. There is no explicit memory mechanism, features being implicitly memorized through a population, and there is no possibility to backtrack to previous states to attempt exploring a different path.

(iii) Agent-based methods: These methods utilize multiple interacting agents, each agent having its own memory and running its own decision-making algorithm [24,25]. Even though traditional agents realize simple decision-making algorithms, e.g., through a set of simple rules in response to specific inputs, it is possible to consider more complex methods, such as each agent running its own synthesis algorithm or population-based evolution. Agents interact with each other by communicating high-quality implementations and features, or implementation steps, which can then be also utilized by the other agents.

Depending on their decision-making procedure, agent-based methods have similar characteristics, like the methods of the previous two categories. Their main advantage is their capacity to simultaneously maintain multiple perspectives about the implementation creation process, e.g., through their local memory, preferences, priorities, etc., and then aggregate these perspectives to improve problem solving. It can be argued that they mimic the implementation creation process of a team (team problem solving) [14,26].

(iv) Cognitive architectures: Cognitive architectures (CAs) mimic the brain activities during problem solving [27–31]. Cognitive Architectures include modules for knowledge representation, knowledge memory, knowledge classification, summarization, comparison, decision-making, prediction, learning, and goal setting. For example, the SOAR CA models cognition-based problem solving [27] using operation selection and application (e.g., state elaboration, operator proposal and evaluation, and decision). Knowledge is procedural if–then rules selected through matching. Learning stores short-cuts to solutions, conditions for applying the rules, and utility updates. ACT-R CA uses multiple symbolic knowledge representations, declarative and procedural information learning, and utility-based decision-making [28]. The EPIC CA matches production rules to the working memory in parallel, followed by the selection of firing rules for multiple goals [29]. The Sigma CA includes mixed symbolic–probabilistic, discrete–continuous representations, knowledge summarization and integration, and inference-based reasoning [30]. The Clarion CA maintains explicit and implicit cognition, each having different representations and processing methods, e.g., rule extraction, generalization, specialization, backpropagation, and reinforcement learning [31]. InnovA is a CA for automated design of electronic circuits [32].

3. Overview of Large Language Models, Prompt Engineering, Retrieval-Augmented Generation, and Reinforcement Learning

3.1. Large Language Models

Large Language Models (LLMs), primarily those built on transformer architectures, have made significant strides in producing coherent, contextually relevant text [33]. They excel at pattern recognition and can generate fluent natural language by leveraging billions of parameters trained on massive corpora [34]. However, their computational principle—self-attention over sequential data—imposes fundamental limitations that hinder their ability to perform rich, open-ended problem-solving tasks.

At the core of these limitations is the reliance on statistical correlations rather than genuine logical understanding. While self-attention excels at identifying relevant tokens in

a sequence, it does not inherently encode hierarchical structures, domain-specific causal rules, or strict logical constraints. This stands in contrast to open-ended problem solving, where the concept space encompasses hierarchical, alternative, and fundamental concepts, and the action space includes complex operations such as feature combination, dynamic adjustment, abstraction, insight generation, and summarization [35]. LLMs struggle to engage these conceptual spaces because they lack mechanisms for hierarchical reasoning, strategic problem decomposition, or flexible reuse of insights [36].

LLMs tend to produce generalized answers aligned with statistical patterns in their training data [37]. Unlike open-ended problem solving, which demands iterative refinement—exploring solution spaces, backtracking, and learning from failures—LLMs typically generate single forward passes [38]. Without internal models of logical inference, memory structures for knowledge accumulation, or explicit strategy formulations, LLMs cannot easily correct their reasoning or adapt based on previous mistakes [39]. This leads to hallucinations, distractions, and inability to build complex, causally grounded explanations.

Some researchers have explored techniques like constraint-based decoding [40], sparse attention mechanisms, adapter layers [41], and memory-augmented transformers. While these approaches enhance performance on certain tasks, they remain add-ons that do not overcome the inherent limitations of attention-based architectures or enable robust open-ended problem solving. The models remain biased toward training data patterns and lack the ability to intentionally search concept space, systematically test hypotheses, or derive new conceptual abstractions [42].

In response to these challenges, methods have emerged to push LLMs toward more sophisticated reasoning and problem-solving behaviors, broadly categorized as prompt engineering, retrieval-augmented generation, and reinforcement learning.

3.2. Prompt Engineering

Prompting techniques utilize carefully constructed input prompts to guide the model's response generation process. Techniques can be grouped into five categories discussed next. Table 1 provides a comprehensive summary of these prompt engineering approaches.

Table 1. Summary of prompt engineering techniques.

Category	Methods	Key Characteristics
Single-Stage Prompting (SSP)	Basic/vanilla prompting	Direct instruction without refinement
	Basic + Term Definitions	Augmented queries with brief definitions
	Annotation Guideline-Based + Error Analysis	Formal entity guidelines + error analysis
Reasoning Strategies	**Linear:** CoT, Complex CoT, ThoT, CoK, CoC, LoT, CoE, chain-of-table	Single step-by-step sequence toward answer
	Branching: Self-Consistency, Contrastive CoT, Fed-SP/DP-SC/COT, ToT, Maieutic	Multiple parallel reasoning paths
	Iterative: PS, PoT, CoS, SCoT, THOR	Step-by-step refinement with feedback loops
Multi-Stage Prompting (MSP)	Ensemble Refinement (ER)	Multiple CoT responses + iterative conditioning
	Auto-CoT	Automatic demonstration construction via clustering
	ReAct	Interleaved reasoning traces with action steps
	Active-Prompt	Adaptive selection of uncertain queries
Knowledge Enhancement	**Example-based:** MathPrompter, analogical reasoning, Synthetic Prompting	High-quality examples and synthesized instances
	Meta-level guidance: self-reflection, S2A, Metacognitive Prompting	Self-monitoring and strategic focusing
Task Decomposition	**Problem breakdown:** Least-to-Most, DecomP, PAL	Hierarchical task decomposition
	Sequential solving: Binder, Dater	Sequential sub-problem execution

(a) Single-stage prompting (SSP): SSP methods directly instruct the model without iterative refinement. Basic/standard/vanilla prompting simply provides a query or instruction to the model, as seen in [43]. Basic with Term Definitions augments queries with brief term definitions to offer extra context, but its impact remains limited since localized definitions may conflict with the model's broader knowledge. Meanwhile, Basic + Annotation Guideline-Based prompting + Error Analysis-Based prompting [44] uses formally defined entity annotation guidelines to specify how clinical terms should be identified and categorized, ensuring clarity in entity recognition. In addition, it incorporates instructions derived from analyzing common model errors, such as addressing ambiguous entity boundaries or redefining prompts for overlapping terms. This strategy significantly improves clinical Named Entity Recognition, with relaxed F1 scores reported as 0.794 for GPT-3.5 and 0.861 for GPT-4 on the MTSamples dataset [45] and 0.676 for GPT-3.5 and 0.736 for GPT-4 on the VAERS dataset [46], demonstrating its effectiveness.

(b) Reasoning strategies: These methods are of three types: linear, branching, and iterative reasoning.

Linear reasoning methods such as Chain-of-Thought (CoT), Complex CoT, Thread-of-Thought (ThoT), Chain-of-Knowledge (CoK), Chain-of-Code (CoC), Logical Thoughts (LoT), Chain-of-Event (CoE), and Chain-of-Table generate a single, step-by-step sequence (chain) of responses toward the final answer. Methods differ in the type of task they target, i.e., code generation, summarization, and logical inference, and in how they refine or represent intermediate steps. CoT shows that using intermediate prompting steps can enhance accuracy, e.g., up to 39% gains in mathematical problem solving [47]. An example of in-context prompt for CoT might be: "*If the problem is 'Calculate 123 × 456,' break it down as (100 + 20 + 3) × 456 and compute step-by-step.*" Complex CoT uses more involved in-context examples, improving performance by as much as 18% on harder tasks [48]. ThoT tackles long or chaotic contexts by *breaking them into manageable parts* (e.g., dividing long passages into sections for sequential summarization) [49], while CoK strategically adapts and consolidates knowledge from multiple sources to ensure coherence and reduce hallucination [50]. CoC specializes in code-oriented reasoning by simulating key code outputs (e.g., predicting intermediate variable states for debugging) [51], whereas LoT integrates logical equivalences and reductio ad absurdum checks to refine reasoning chains (e.g., validating statements by identifying contradictions in their negations) [52]. CoE handles summarization by extracting, generalizing, filtering, and integrating key events (e.g., pinpointing main events from news articles) [53], and Chain-of-Table extends CoT techniques to tabular data, dynamically applying transformations like filtering or aggregation to generate coherent answers [54].

Branching reasoning methods, like Self-Consistency, Contrastive CoT (or Contrastive Self-Consistency), Federated Same/Different Parameter Self-Consistency/CoT (Fed-SP/DP-SC/COT), Tree-of-Thoughts, and Maieutic prompting, explore multiple possible reasoning paths in parallel. However, they vary in how they sample or fuse paths, some relying on consensus votes and others on dynamic adaptation or tree-based elimination. Self-Consistency, for instance, samples diverse solution paths and selects the most consistent final answer, achieving gains of over 11% on math tasks [55]. Contrastive CoT incorporates both correct and incorrect in-context examples to broaden the model's understanding, improving performance by over 10% compared to standard CoT [56]. Fed-SP-SC leverages paraphrased queries to crowdsource additional hints [57], while ToT maintains a tree of partial solutions and systematically explores them with breadth-first or depth-first strategies, offering up to 65% higher success rates than CoT on challenging math tasks [58]. Maieutic prompting likewise generates a tree of propositions to reconcile contradictory statements, surpassing linear methods by 20% on common-sense benchmarks [59].

Iterative reasoning approaches, such as Plan-and-Solve (PS), Program-of-Thoughts (PoT), Chain-of-Symbol (CoS), Structured Chain-of-Thought (SCoT), and Three-Hop Reasoning (THOR), refine solutions step by step, often bypassing intermediate outputs back into the model to enhance accuracy. PS explicitly decomposes tasks into planning and execution phases, where the planning phase structures the problem into smaller sub-tasks, and the execution phase solves them sequentially. This reduces semantic and calculation errors, outperforming Chain-of-Thought (CoT) prompting by up to 5% [60]. PoT enhances performance by separating reasoning from computation: the model generates programmatic solutions executed by a Python interpreter, achieving up to 12% accuracy gains in numerical and QA tasks [61]. CoS encodes spatial and symbolic relationships using concise symbolic representations, which improves reasoning in spatial tasks by up to 60.8% [62]. SCoT introduces structured reasoning through program-like branching and looping, significantly improving code generation accuracy by up to 13.79% [63]. Finally, THOR tackles emotion and sentiment analysis through a three-stage approach: aspect identification, opinion analysis, and polarity inference. This structured method achieves superior performance compared to previous supervised and zero-shot models [64]. These approaches exemplify the power of iterative methods in breaking complex problems into manageable components, thereby reducing errors and improving overall performance.

(c) Multi-Stage Prompting (MSP): MSP techniques rely on iterative feedback loops or ensemble strategies. MSP methods systematically refine outputs and incorporate multiple response paths, e.g., through voting or iterative analysis, to yield more robust and accurate solutions, particularly in domains requiring deeper reasoning or tailored task adaptation. Ensemble Refinement (ER) [65] builds on Chain-of-Thought (CoT) and Self-Consistency by generating multiple CoT-based responses at high temperature (introducing diversity) and then iteratively conditioning on generated responses to produce a more coherent and accurate output, leveraging insights from the strengths and weaknesses of initial explanations and majority voting. Auto-CoT [66] constructs demonstrations automatically by clustering queries from a dataset and generating reasoning chains for representative queries using zero-shot CoT. Clustering is achieved by partitioning questions into groups based on semantic similarity, ensuring that representative queries capture the diversity of the dataset. ReAct [67] interleaves reasoning traces—thought processes that explain intermediate steps—with action steps that execute operations, enabling superior performance in complex tasks by seamlessly combining reasoning and action. Moreover, Active-Prompt [68] adaptively selects the most uncertain training queries, identified via confidence metrics like entropy or variance, for human annotation, boosting few-shot learning performance by focusing on areas with the highest uncertainty.

(d) Knowledge Enhancement: These approaches use high-quality examples and strategic self-monitoring to improve LLM performance. They pertain to two types, example-based and meta-level guidance methods.

Example-based methods leverage auxiliary examples or synthesized instances to guide the response creation process of LLMs. MathPrompter [69] focuses on creating a symbolic template of the given mathematical query, solving it analytically or via Python, and then validating the derived solution with random variable substitutions before finalizing the answer. The approach boosts accuracy from 78.7% to 92.5%. analogical reasoning [70] prompts LLMs to generate and solve similar examples before addressing the main problem, resulting in a 4% average accuracy gain across various tasks. Synthetic Prompting [71] involves a backward step, where a new query is generated from a self-constructed reasoning chain, and a forward step, where this query is re-solved; this strategy selects the most complex examples for few-shot prompts, leading to up to 15.6% absolute improvements in mathematical problem solving, common-sense reasoning, and logical reasoning.

Meta-Level Guidance (MLG) methods enhance LLMs by promoting self-reflection and focusing on pertinent information, thereby reducing errors. Self-reflection involves the model evaluating its own outputs to identify and correct mistakes, leading to improved performance. For example, in translation tasks, self-reflection enables LLMs to retrieve bilingual knowledge, facilitating the generation of higher-quality translations. Focusing is achieved through techniques like System 2 Attention (S2A) [72], which filters out irrelevant content by prompting the model to regenerate the context to include only essential information before producing a final response. This two-step approach enhances reasoning by concentrating on relevant details, thereby improving accuracy. S2A has been shown to outperform basic prompting methods, including Chain-of-Thought (CoT) and instructed prompting, particularly on truthfulness-oriented datasets. Metacognitive Prompting (MP) [73] introduces a five-stage process to further enhance LLM performance: (1) Comprehension: The model attempts to understand the input, ensuring clarity before proceeding. (2) Preliminary Judgment: An initial assessment is made based on the understood information. (3) Critical Evaluation: The initial judgment is scrutinized, considering alternative perspectives and potential errors. (4) Final Decision with Explanation: A conclusive decision is reached, accompanied by a rationale to support it. (5) Self-Assessment of Confidence: The model evaluates its confidence in the final decision, reflecting on the reasoning process. This structured approach enables LLMs to perform consistently better than methods like CoT and Program Synthesis (PS) across various natural language processing tasks, including paraphrasing, natural language inference, and named entity recognition.

(e) Task Decomposition: These approaches break down complex tasks into smaller steps but vary in how they orchestrate and execute the sub-problems. They include problem breakdown and sequential solving methods.

Problem breakdown approaches include the Least-to-Most method [74], which addresses the challenge of Chain-of-Thought (CoT) failing on problems more difficult than its exemplars by first prompting the LLM to decompose a query into sub-problems and then solving them sequentially, demonstrating notable improvements over CoT and basic prompting on tasks like commonsense reasoning and mathematical problem solving. The decompositions are characterized by their hierarchical structure, breaking down complex problems into simpler, manageable sub-tasks that build upon each other to facilitate step-by-step reasoning. Decomposed Prompting (DecomP) breaks complex tasks into simpler sub-tasks, each handled with tailored prompts or external tools, ensuring efficient and accurate execution. For instance, the task "Concatenate the first letters of words in 'Jack Ryan'" is decomposed into extracting words, finding their first letters, and concatenating them [75]. DecomP leverages modular decomposers to partition problems hierarchically or recursively, assigning sub-tasks to specialized LLMs or APIs. This approach achieves a 25% improvement over CoT and Least-to-Most methods in commonsense reasoning. Program-Aided Language models (PAL) [76] further leverage interleaved natural language and programmatic steps to enable Python-based execution of the reasoning process, surpassing CoT and basic methods for mathematical and commonsense tasks.

Sequential solving includes methods like Binder and Dater algorithms. Binder [77] integrates neural and symbolic parts by using an LLM both as a parser and executor for natural language queries, leveraging programming languages like Python or SQL for structured execution. Binding is achieved through a unified API that enables the LLM to generate, interpret, and execute code using a few in-context examples, leading to higher accuracy on table-based tasks compared to fine-tuned approaches. Dater [78] focuses on few-shot table reasoning by splitting a large table into relevant sub-tables, translating complex queries into SQL sub-queries, and combining partial outcomes into a final solution. These three steps aim to systematically extract meaningful data, execute precise operations,

and integrate results to address complex queries, outperforming fine-tuned methods by at least 2% on Table-Based Truthfulness and 1% on Table-Based QA, and surpassing Binder on these tasks.

3.3. Retrieval-Augmented Generation

Retrieval-augmented generation (RAG) addresses LLMs' lack of persistent memory and factual grounding by integrating external knowledge sources into the generation process [79]. Instead of relying solely on learned parameters, RAG systems retrieve relevant documents, facts, or structured data at inference time, reducing hallucinations and ensuring reasoning steps reference accurate, up-to-date information [80]. RAG advancements span healthcare, finance, education, and scientific research, addressing challenges in reasoning, problem-solving, and knowledge integration. This review categorizes these advancements into four areas: task-specific and schema-based techniques, self-aware and adaptive mechanisms, long-term memory integration, and multi-hop and multi-modal reasoning. Table 2 provides a comprehensive summary of these RAG approaches.

Table 2. Summary of retrieval-augmented generation techniques.

Category	Methods	Key Characteristics
Task-Specific and Schema-Based Retrieval	**Schema-based:** SBI-RAG, KRAGEN	Predefined structures with knowledge graphs, graph-of-thoughts reasoning, visualization tools for explainability
	Data-driven: GRAM, TableRAG	Dynamic schema alignment via hierarchical classification, tabular reasoning with query expansion
Self-Aware and Adaptive Retrieval	SeaKR	Uncertainty detection via internal state inconsistencies, confidence-based retrieval triggering, snippet re-ranking
	Self-RAG	Token-based strategies with Retrieve and Critique tokens, iterative refinement cycles
	Speculative RAG	Two-stage process with smaller and larger models, clustered document subsets, parallel draft generation
	SimRAG	Self-training with pseudo-labeled data for domain-specific adaptation
	CR-Planner	Critic models with MCTS for sub-goal evaluation and execution path selection
	SeRTS	Tree search framework with MCTS and PPO for snippet integration and query refinement
Long-Term Memory Integration	HippoRAG	Knowledge graphs with Personalized PageRank for multi-hop reasoning
	HybridRAG	Combines structured KGs with vector-based retrieval for unstructured data
	GRAG, SimGRAG	K-hop ego-graphs with soft pruning, similar subgraph identification
	MemLong, HAT, MemoRAG	Compressed memory buffers, hierarchical aggregate trees, decoupled memory updates
	Pistis-RAG	Multistage processing with human feedback adaptation
Multi-Hop and Multi-Modal Reasoning	**Multi-hop:** METRAG, RAG-Star, KRAGEN	Similarity-oriented retrieval, MCTS with verification, graph-of-thoughts decomposition
	Multi-modal: M3DocRAG, VisRAG, OmniSearch	Visual embeddings, vision–language models, self-adaptive planning for diverse data formats
Self-Critique Methods	CoVe	Four-step verification process with dynamic question generation and bias reduction
	VE	Self-consistency with majority voting, validated evidence integration

(a) Task-Specific and Schema-Based Retrieval (TSR): TSR approaches employ structured methods for domain-specific challenges in mathematics and knowledge-intensive tasks, categorized into schema-based and data-driven techniques.

Schema-based techniques use predefined structures to guide reasoning. Schema-Based Instruction RAG (SBI-RAG) [81] formalizes schemas to represent problem structures through knowledge graphs that map domain-specific concepts. The Knowledge Graph-Enhanced RAG Framework (KRAGEN) [82] employs graph-of-thoughts reasoning to decompose tasks into sub-problems, retrieve relevant knowledge, and synthesize solutions. KRAGEN integrates visualization tools for explainability, allowing users to trace sub-

problem decomposition and validate reasoning processes. However, reliance on predefined templates limits performance on novel problems.

Data-driven retrieval methods extract insights without predefined schemas. Generative Retrieval-Augmented Matching (GRAM) [83] uses hierarchical classification for dynamic schema alignment, performing coarse-grained matching followed by fine-grained context-based refinement. GRAM adapts to evolving query patterns with minimal labeled data but struggles in noisy environments. TableRAG [84] reasons over tabular data through query expansion, schema retrieval, and cell-level reasoning, though query expansion introduces latency unsuitable for real-time tasks.

(b) Self-Aware and Adaptive Retrieval: Self-aware RAG frameworks address ambiguous queries, conflicting information, and knowledge gaps through adaptive mechanisms.

SeaKR [85] detects uncertainty through internal state inconsistencies, triggering retrieval when confidence is low and re-ranking snippets to reduce uncertainty. While effective for uncertainty detection, it struggles with domain-specific tasks and may trigger unnecessary retrievals for complex but unambiguous queries.

Self-Reflective RAG (Self-RAG) [86] uses token-based strategies for iterative refinement. Retrieve tokens assess query complexity to determine retrieval needs, while Critique tokens evaluate responses across relevance, support, and usefulness dimensions. Through iterative retrieval-critique cycles, Self-RAG adapts responses to retrieved knowledge, though this increases latency and may overfit to evaluation criteria.

Speculative RAG [87] employs a two-stage process: a smaller model generates diverse draft responses from clustered document subsets, then a larger model evaluates drafts via conditional generation probabilities. This achieves 12.97% accuracy improvement and 51% latency reduction on PubHealth, though it lacks uncertainty detection mechanisms and may miss nuances in dynamic queries.

SimRAG [88] fine-tunes RAG for specialized domains through self-training with pseudo-labeled data generated from unlabeled corpora. While addressing domain-specific knowledge gaps, it risks overfitting and lacks explicit uncertainty detection.

CR-Planner [89] uses critic models trained via Monte Carlo Tree Search to evaluate sub-goals and execution paths, decomposing tasks for clarity. However, MCTS incurs high computational costs, and domain-specific critics limit generalizability.

Self-Rewarding Tree Search (SeRTS) [90] frames retrieval as tree search, using MCTS with upper confidence bounds to select nodes and Proximal Policy Optimization to refine strategies. While effective for biomedical retrieval, tree search becomes computationally expensive for complex queries.

(c) Long-Term Memory for Knowledge Retrieval: Long-term memory integration addresses limitations of query-specific retrieval: poor context retention, limited reusability, and inability to accommodate human feedback [91].

HippoRAG [92] organizes information using knowledge graphs (KGs) where entities are nodes and relationships are edges, constructed via Open Information Extraction. Personalized PageRank retrieves relevant subgraphs for queries, supporting multi-hop reasoning by combining relationships across nodes. HippoRAG achieves 20% improvement on multi-hop benchmarks like 2WikiMultiHopQA [93] and MuSiQue [94].

Long-term memory systems can be characterized across five dimensions: (i) KG types and management, (ii) relevant information identification, (iii) knowledge utilization mechanisms, (iv) memory organization, and (v) response integration into KGs.

For dimension (i), HybridRAG combines structured KGs with vector-based retrieval for unstructured data, enhancing retrieval in domains like finance requiring both precise definitions and broad context [95]. For dimension (ii), GRAG retrieves k-hop ego-graphs

with soft pruning [96], while SimGRAG identifies similar subgraphs using recurring patterns for consistency [97].

For dimension (iv), MemLong [98] maintains a compressed memory buffer with dynamic caching, handling up to 65,000 tokens while prioritizing frequently accessed information. HAT [99] organizes dialogue history into hierarchical aggregate trees for recursive aggregation. MemoRAG [100] decouples memory updates from retrieval, using draft answers as clues for precise retrieval and processing up to 600K tokens through token compression.

For dimension (v), Pistis-RAG [101] adapts to human feedback through multistage processing (matching, pre-ranking, ranking, reasoning, aggregating), achieving 9.3% improvement on MMLU, though continuous feedback may introduce variability.

(d) Multi-Hop and Multi-Modal Reasoning Retrieval: Multi-hop reasoning connects information across multiple steps for coherent answers, while multi-modal retrieval synthesizes data from images, text, audio, and video.

Multi-layered Thoughts Enhanced RAG (METRAG) [102] employs four components: similarity-oriented retrieval identifies relevant documents, utility assessment evaluates usefulness via task relevance and completeness metrics, task-adaptive summarization reduces redundancy, and contextual reasoning integrates information for nuanced outputs.

RAG-Star [103] integrates Monte Carlo Tree Search with retrieval-augmented verification, balancing exploration and exploitation through random simulations. Query- and answer-aware reward modeling refines reasoning trajectories for multi-step logical inference.

KRAGEN [82] uses Graph-of-Thoughts to decompose queries into subproblems represented as dynamically constructed graphs. Vectorized embeddings retrieve relevant data from domain-specific knowledge graphs, synthesizing results while preserving dependencies. However, multi-hop techniques rely heavily on underlying knowledge source quality.

Multi-modal extensions handle diverse data formats. M3DocRAG [104] encodes document images into visual embeddings, retrieves pages via multi-modal models, and generates answers for both single-hop and multi-hop queries. VisRAG [105] embeds documents directly as images using vision–language models, avoiding text parsing loss while retaining structure and context. OmniSearch [106] employs self-adaptive planning to decompose multi-modal queries into sub-question chains, adapting retrieval strategies based on query characteristics and temporal factors.

(e) Self-Critique Methods: Self-critique methodologies systematically verify and refine outputs to enhance reliability and factual accuracy in RAG systems.

Chain-of-Verification (CoVe) [107] follows four steps: generating an initial response, formulating verification questions to fact-check the response, independently answering these questions to reduce bias, and revising the answer using validated information. The model dynamically generates verification questions by conditioning on the query and baseline response, cross-referencing with existing knowledge sources. CoVe achieves over 10% performance gains compared to standard prompting and Chain-of-Thought methods.

Verify-and-Edit (VE) [108] identifies weak or contradictory reasoning through self-consistency by generating multiple solutions and using majority voting to pinpoint uncertainty. VE integrates validated evidence from trusted sources, demonstrating 9.5% gains in multi-hop reasoning and 25.6% improvement in truthfulness evaluations.

3.4. Reinforcement Learning

Reinforcement learning (RL) refines LLM behavior through iterative feedback and reward signals from human feedback, automated metrics, or pre-trained reward models [109]. RL comprises six components: agent, environment, state, action, reward, and policy [110]. For LLM fine-tuning, the LLM represents the policy, the textual sequence is the state,

and token generation is the action. After generating a complete sequence, a reward assesses output quality, either training a reward model or directly guiding alignment. RL methods divide into model-based and model-free approaches. Table 3 provides a comprehensive summary of these RL techniques.

Table 3. Summary of reinforcement learning techniques for LLMs.

Category	Methods	Key Characteristics
Model-Based RL		
RLHF	InstructGPT, trlX, UltraFeedback	Three-stage process: SFT, reward model training, PPO fine-tuning; addresses length bias; resource-intensive
	Skywork-Reward, TÜLU-V2-mix	Curated datasets of ranked examples for reward model training
	Tool-augmented RM, Pairwise PPO	External resource integration, pairwise feedback for preference learning
RLAIF	UltraFeedback, Magpie, HelpSteer2	AI-generated preference labels, self-synthesis methods, scalable but risks bias propagation
	Eureka, Self-Rewarding LMs	Self-supervised feedback, iterative refinement through self-evaluation
Exploration	ExploRLLM, Soft RLLF, LLM + Exp, Guided Pretraining RL	LLM-guided exploration with high-level plans, logical feedback, dual-LLM frameworks, structured trajectories
Model-Free RL		
DPO	DPO, DPOP, Iterative DPO	Bypasses reward model, direct parameter optimization, margin-based terms, continuous policy updates
	β-DPO, Stepwise DPO	Adaptive regularization, incremental updates with intermediate reference models
IPO	IPO	Avoids nonlinear transformations, linear objective function, robust for deterministic feedback
Actor-Critic	PACE, HDFlow, DQO	Actor generates responses, critic evaluates; KL-regularization; fast/slow thinking; MDP formulation

(a) Model-based RL Approaches: The methods in this category can be grouped into three categories, RLHF, RLAIF, and exploration, which are discussed next.

Reinforcement Learning from Human Feedback (RLHF) incorporates reward signals from human evaluations through three stages: supervised fine-tuning (SFT) with labeled datasets, training a reward model (RM) from human-evaluated outputs, and policy fine-tuning using Proximal Policy Optimization (PPO) [111]. InstructGPT [112] exemplifies RLHF's effectiveness in instruction adherence. These methods address challenges like length bias [113,114], while frameworks like trlX [115] and datasets like UltraFeedback [116] scale RLHF to tasks including summarization, translation, and dialogue generation.

PPO iteratively adjusts model weights to maximize expected rewards. Reward model training relies on human feedback through curated datasets of ranked examples, as in Skywork-Reward [117] and TÜLU-V2-mix [118]. Tool-augmented reward modeling [119] integrates external resources like calculators, while generative reward models use synthetic preferences to reduce human feedback dependence. Pairwise feedback pipelines [120] improve preference learning by comparing response pairs. However, RLHF remains resource-intensive and risks over-optimization, where models exploit reward function weaknesses rather than achieving genuine alignment [121,122].

RL from AI Feedback (RLAIF) replaces human evaluators with AI systems for better scalability and consistency [123]. Reward models are trained using LLM-generated preference labels transformed via softmax and optimized through cross-entropy loss [124].

Approaches include large-scale datasets like UltraFeedback with GPT-4 annotations [116], self-synthesis methods like Magpie [125], and permissively licensed datasets like HelpSteer2 [126]. LLMs can also function directly as reward functions [127,128]. Self-supervised mechanisms like Eureka [129] and self-rewarding systems [130,131] enable iterative refinement through self-generated feedback. However, RLAIF risks propagating AI biases, creating feedback loops that constrain diversity and limit generalization [116,129,131].

Exploration techniques balance seeking new information with exploiting current knowledge [132]. Traditional approaches like epsilon-greedy [133,134] and Boltzmann exploration [135] introduce randomness but slow convergence. Recent methods leverage LLMs for strategic exploration. ExploRLLM [136] combines LLM-generated high-level plans with affordance-based policies for action execution, improving efficiency in structured environments but struggling with dynamic domains [137]. Soft RLLF [138] integrates natural language as logical feedback for reasoning tasks, though it is optimized for structured rather than creative problems. LLM + Exp [139] employs dual LLMs to analyze action–reward trajectories and adjust action probabilities, excelling in structured environments but facing scalability issues in unpredictable tasks. Guided Pretraining RL [127] uses LLM-generated structured trajectories to pretrain agents, improving sample efficiency but limiting generalization to variable environments.

(b) Model-Free Approaches: These methods can be grouped into three categories, DPO, IPO, and actor–critic, which are discussed next.

Direct Preference Optimization (DPO) simplifies RLHF by directly optimizing LLM parameters using preference data, bypassing reward model training [140]. DPO uses preference loss functions on paired human preferences. Extensions include DPOP [141], which introduces margin-based terms to prevent rewarding both preferred and disfavored outputs; Iterative DPO [131], which mitigates distribution shifts through continuous policy updates; β-DPO [142], which adaptively tunes regularization based on data quality; and Stepwise DPO [143], which performs incremental updates with stronger intermediate reference models. DPO excels in structured problem-solving by incorporating human preferences without complex RL training [140]. However, it is sensitive to distribution shifts and can introduce label noise in creative tasks [144,145].

Identity Preference Optimization (IPO) [146] addresses overfitting in RLHF and DPO by directly optimizing preferences without nonlinear transformations. Unlike the Bradley–Terry model [147], IPO uses a linear objective function with KL divergence regularization, formulating a squared loss over preferred and less preferred output pairs. This approach is robust for deterministic feedback scenarios and outperforms DPO [148,149]. However, IPO's reliance on static preference distributions limits adaptability, and its sensitivity to noise reduces robustness in complex environments [150].

Actor–critic methods like Advantage Actor–Critic (A2C) and Deep Deterministic Policy Gradient (DDPG) optimize LLM prompts through iterative actor–critic loops. Prompt Actor–Critic Editing (PACE) [151] uses an actor to generate responses from prompts and inputs, while a critic evaluates relevance and accuracy against objectives. KL-regularization [152] balances prompt fidelity with task-specific improvements. However, actor–critic methods assume well-structured feedback, challenging for sparse or noisy signals. Recent work addresses this: HDFlow [153] combines fast and slow thinking modes for complex reasoning, while Direct Q-function Optimization (DQO) [154] formulates response generation as a Markov Decision Process, parameterizing the Q-function within the LLM to learn from offline data including unbalanced samples, improving multi-step reasoning.

4. A Novel LLM-Based Cognitive Architecture

This section discusses devising a Cognitive Architecture (CA) for problem solving using the features of an LLM. The LLM is a comprehensive but generic pool of knowledge. The knowledge organization (that emerges through LLM learning) is unknown but likely to be determined by the training procedure, thus it is not customized to the actual problem to be solved or to the pursued problem-solving procedure. Hence, the relevant knowledge is likely to be scattered throughout the LLM space. Figure 1 details this aspect by referring to the LLM knowledge space, problem space, and solution space. The problem space is defined by the related problem variables and requirements, but the contiguous space is projected into fragmented sub-spaces, e.g., $Sub\text{-}space_1$ and $Sub\text{-}space_2$ in the figure. This can be explained by the fact that the LLM knowledge space was trained on data (i.e., problems) that covered only some of the problem requirements, if the problems were similar but not identical to the current problem. Similarly, the solution space is also projected on the LLM knowledge space as fragmented sub-spaces, like $Sub\text{-}space_a$ and $Sub\text{-}space_b$. Figure 2 illustrates the fragmentation, where darker areas indicate areas with more relevant information.

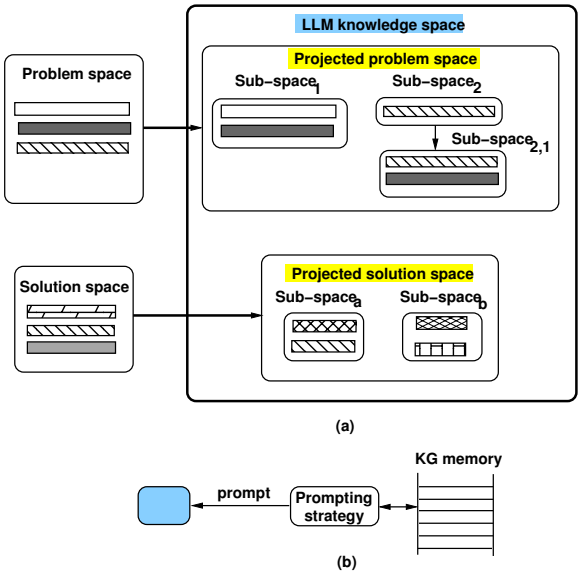


Figure 1. (**a**) Projection of the problem space and solution space onto the LLM knowledge space and (**b**) implementation of repeated prompting.

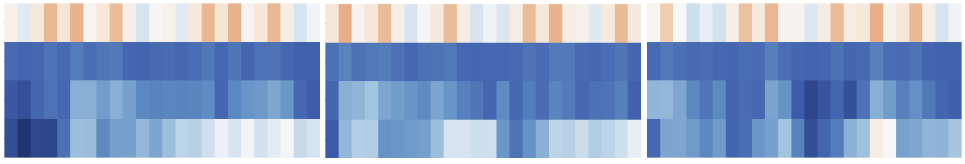


Figure 2. (**Left**) Mean activation magnitudes across tokens and layers after the initial prompt. (**Middle**) Mean activation magnitudes across tokens and layers after the intermediate prompt. (**Right**) Mean activation magnitudes across tokens and layers after the final prompt (stabilized).

The fragmentation of the projections onto the LLM space creates a situation in which simple LLM prompting to create a solution does not offer the best responses, as the sub-spaces correspond only to some of the problem constraints. This work suggests two elements to address this aspect: the stability criterion to describe the repeated prompting of the LLM to improve the LLM responses and using a knowledge graph as a memory to implement the repeated prompting process. Figure 2 illustrates the evolution of layer-wise activation magnitudes across tokens over three iterations, reflecting this process.

Stability criterion: Prompting should be repeated until $\frac{\delta Q(r)}{\delta c}, \frac{\delta r}{\delta c} \approx 0$, where r is the LLM response (output) for prompt p and context c, and Q describes the quality of the response.

Proof. As $Q(r) = Q(F(c,p))$, where F is the LLM's function, the response is stable when $\frac{\delta Q(F(c,p))}{\delta c} \approx 0$. Hence, $\frac{\delta Q(F(c,p))}{\delta c} = \frac{\delta Q}{\delta c}(F(c,p))\frac{\delta F(c,p)}{\delta c} = 0$, which justifies the two conditions. □

The criterion indicates that LLM prompting should incrementally change the context until the two conditions are met, thus the obtained, optimized response is stable. Figure 1b depicts the structure for iterative prompting to implement the stability criterion. It follows the RAG concept, in which a knowledge graph (KG) memory keeps track of the contexts used to create new prompts through a prompting strategy method. The initial prompt (Figure 2(left)) shows dispersed activations across many layers and token positions, reflecting the model's initial uncertainty and projection into multiple sub-spaces. After injecting context from the first response stored in the KG, the intermediate prompt (Figure 2(middle)) shows concentrated activations in a narrower band of layers, indicating a focus on a more relevant problem sub-space. With further context refinement, the final prompt (Figure 2(right)) produces activations converging sharply in those layers and tokens associated with the correct solution manifold, with negligible change confirming that the stability criterion has been met.

4.1. Problem-Solving Activities

The following activities are part of any problem-solving strategy, hence should be supported by the proposed CA too:

1. *Divide and combine*: The activity partitions a problem into sub-problems and then provides ways to combine (integrate) the implementation for the sub-problems. Task decomposition methods in LLM prompting [74,75] can produce certain decompositions, especially in situations for which the sub-problems are less coupled. However, many design problems are strongly coupled, so that even though there are specialized modules to implement a certain function, their operation and performance are tightly related. Then, decomposition requires not only a static problem partitioning based on the items in the prompts (i.e., words) but also the interpretation of a sub-problem within the context set up by the interpretations of the other sub-problems.
   LLM fine-tuning through RL is likely infeasible due to the large space of possible decompositions in real life. A mechanism is also needed to track the analyzed decompositions, so that the information can be used to improve future divisions. This capability is absent in current methods.
   The stability criterion is instantiated as follows for problem division into sub-problems:
   $$Q(F(c_1 \cup \ldots c_k, p_1 \cup \ldots p_k)) < Q(F(c_1, p_1) \cup \ldots F(c_k, p_k))$$
   The relationship states that the quality obtained by separately prompting the sub-problems for the k contexts and prompts is higher than the quality obtained if the entire problem is prompted using a single (overall) context and prompt.
2. *Elaboration*: Similar to the CA in [32], this paper considers that devising an implementation for a problem specification utilizes the five strategies shown in Figure 3. The five strategies are detailed later in this section. The problem-solving process is a mixture of the five strategies.
   Executing the five strategies requires devising additional methods for detailing alternatives, predicting the effectiveness of each alternative in the context of a partial implementation, assigning a priority (preference) to each alternative, and incorporating the alternative into the partial implementation. As detailed later in the paper, the proposed approach used a RAG method for the five strategies.

3. *Implementation assessment*: LLMs can be used for two kinds of performance assessments. Qualitative assessment, including comparing implementations, such as pairs of circuit designs, can be obtained by prompting traditional LLMs. CoT prompting can be used to obtain performance assessment at a finer granularity. RLHF can fine-tune assessment by adding human feedback about the quality of the implementations [116]. Moreover, self-critique methods can improve the correctness and completeness of the LLM responses, like self-consistency and cross-referencing methods in VE [108]. A second approach uses datasets of characterized implementations to train an LLM, similar to exploration techniques in RL [134,135]. Then, the generalization capacity of LLMs are used to quantitatively predict the performance parameters of a new implementation. Nevertheless, the two approaches do not scale beyond the samples used in training an LLM, including situations in which a new implementation uses a nonlinear combination of the features of different implementations. There is no mechanism similar to setting-up precise physical models of an implementation, so that the models can be solved to produce quantitative performance assessment, like in traditional automated implementation creation methods.
4. *Memory and learning*: Similar to using long-term memory for knowledge retrieval in RAG, memory systems are needed by the CA for learning to store associations, like kernel features, their most relevant implementations fragments, and their performance values or between high-level features and their detailed elaborations, the causal relationships of main features and performance attributes, and elaboration sequences that produced high-quality implementations. Similar to schema-based retrieval, memory cuing must solve semantic ambiguities. The memory of the proposed CA stores knowledge graphs, which are schemata, similar to those presented in the cognitive psychology literature [155].
5. *Adaptive process*: It includes the sequence of automated activities performed to create an implementation. It requires devising new means to predict the expected outcomes of the activities, selecting and adapting an activity to the current context, understanding the degree to which the sequence advances towards creating an implementation, and learning new knowledge available during the process. Also, when addressing collaboration between humans and LLMs to tackle unexpected challenges, such as handling zero-day attacks, the process necessitates reasoning, understanding of prior instructions, and intuitive decision-making within the context of new parameters and constraints. To automate this process, exploring reasoning techniques is required, including deductive reasoning, inductive reasoning, analogical reasoning, commonsense reasoning, tree-of-thoughts, multiple chains of thought, causal reasoning, heuristic reasoning, and symbolic reasoning. Among these, the primary human thought process often involves mapping the current problem to a previously encountered one or identifying similarities with analogous problems, like in analogical reasoning. Consequently, an effective approach to problem modeling could involve neurosymbolic representations that allow LLMs to dynamically learn and adapt in real time. Techniques such as grokking [156], which enable models to discover relationships and patterns through iterative refinement, and masked LLMs are promising methods to achieve this goal. These approaches empower the model to derive connections on the fly, effectively merging learned representations with reasoning capabilities.

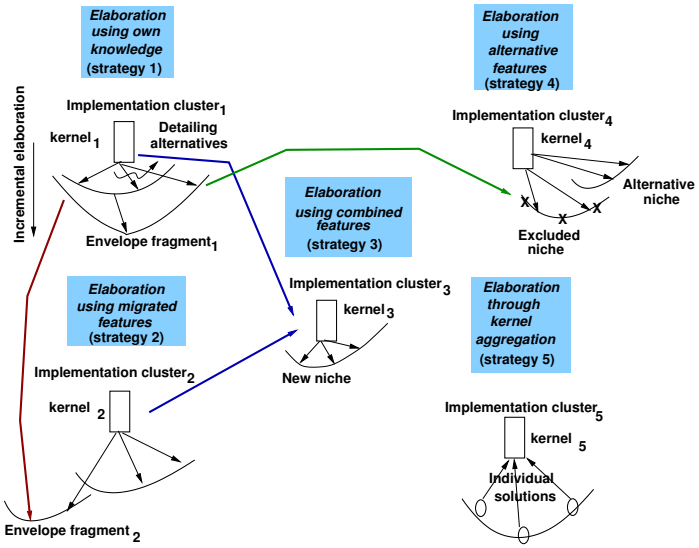


Figure 3. Five strategies for automated implementation creation.

4.2. LLM-Based Cognitive Architecture

This paper focuses on using an LLM to implement a CA based on the five problem-solving strategies depicted in Figure 3, including the activities of elaboration and memory and learning. The other activities are left for future work.

Each strategy starts from a *kernel*, which is the invariant (constant) set of features used in the problem-solving process. The process creates a solution cluster corresponding to the kernel features, e.g., each implementation in the cluster includes the features. Implementations are created through implementation elaboration by exploring a sequence of detailing alternatives. For example, the principle of the bubble sort algorithm can be described as repeatedly comparing the adjacent values of an array and swapping them if they are in the wrong order until no more value swapping are needed. The kernel features are (i) the values of an array, (ii) the swapping of adjacent values if they are in the wrong order, and (iii) the repetition of the process until no more swapping are needed. The corresponding cluster includes all implementations obtained by elaborating the three kernel features.

The five strategies used by the CA are as follows [32]:

- Strategy 1 describes the elaboration process in which each kernel is elaborated without changing the kernel. A set of detailing alternatives can be used for each elaboration step to produce an implementation envelope. The envelopes are incrementally elaborated top-down until the final implementation is created.
- Strategy 2 represents the process, which, in addition to the elaboration steps of Strategy 1, also uses elaboration results corresponding to a different implementation cluster. Figure 3 shows the use of features from Implementation cluster 1 (red arrow in the figure) to build the implementations of Implementation cluster 2. Hence, the subsequent solution include top-down elaboration of all kernel features and the features adopted from another cluster.
- Strategy 3 uses a kernel that combines top-down kernel features from two different implementation clusters. The blue arrows in Figure 3 illustrates the combination.
- Strategy 4 presents an elaboration process in which the selected detailing alternatives are excluded from the elaboration steps used for building other implementation clusters. It represents the excluded niche in Figure 3 (green arrow).
- Strategy 5 creates a bottom-up kernel by identifying and generalizing the features of individual implementations. The individual implementations were produced through less-structured methods, like, for example, experimental trial and error.

The CA creates kernels either by assembling the features likely to address the problem requirements and then elaborating them top-down or in a bottom-up process as detailed in Strategy 5. Separate kernels can be created for different sub-problems followed by integrating them into a single kernel and elaborating it or separately elaborating each kernel and integrating their implementations. Ideas on LLM self-reflection and focus on the main information [72] can help identify the features to be included in a kernel. Figure 4a,b illustrate two possible kernels. Figure 4a shows a kernel for a decision-making action to change the position of data (e.g., for sorting), computing new values, and finding relations among data. Figure 4b shows the kernel for sequences of decisions.

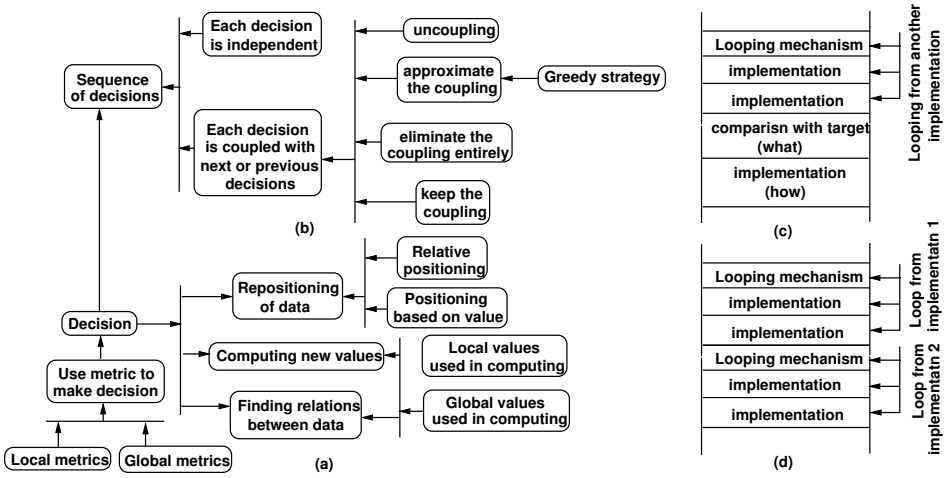


Figure 4. Illustrations of kernels and strategies: (**a**) decision-making actions, (**b**) sequences of decisions, (**c**) KG RAG storing kernel elaborations (Strategy 2), and (**d**) KG RAG combining conceptual ideas with distinct looping mechanisms (Strategy 3).

Five strategies for automated implementation creation: Finding kernels, e.g., the invariant features present in all implementations pertaining to a cluster, remains mostly a manual process, which is a limitation of the current CA. Methods similar to RLHF [111] can help retrieving similar features, but their scalability is likely low. Moreover, combining features from different kernels to generate a new kernel (Strategy 3) has not been extensively studied by current LLM methods, but it has been a traditional research area in cognitive psychology [22]. The combination of features needs a way to predict the expected performance at a high level (possibly a qualitative evaluation), which can be offered to some degree by LLM, similar to the use of LLM to solve ambiguities [83,84]. However, it is likely that the current methods are insufficient for this purpose.

The realization of the five strategies is presented next.

Strategy 1. This strategy realizes top-down elaboration, in which conceptual ideas as well as any elaborations expressed by a team during its verbal discussions are converted into an implementation. The algorithm for LLM prompting to implement Strategy 1, adapted to this context, is shown in Algorithm 1.

The algorithm identifies the feature sets FS (problem requirements) and DFS (solution features from the discussion). LLMs can be prompted to extract these features from natural language discussions, identify inconsistencies, and generate a logical sequence of operations to produce executable code. To align with the stability criterion, prompting iterates until the response quality stabilizes ($\frac{\delta Q(r)}{\delta c}, \frac{\delta r}{\delta c} \approx 0$) by checking that there are no inconsistencies between the verbal discussion, the LLM-generated solution, and problem description. Then, as part of solution elaboration, the LLM is first prompted to find inconsistencies between problem and solution features. The LLM is next prompted using the dialog as a context to find the relationships between variables as well as the sequence between relationships.

The obtained implementation is *SOLUTION*. The next two prompts ask for inconsistencies between *SOLUTION* and between it and the problem description.

Algorithm 1 LLM-based implementation of Strategy 1

Require: problem description, discussion
Ensure: *SOLUTION* (e.g., generated code)
1: *FS* = set of features into which the problem is decomposed;
2: *DFS* = set of features of the discussed solution;
3: **while** there are any inconsistencies **do**
4: prompt LLM to find inconsistencies between sets *FS* and *DFS*;
5: prompt LLM to find the main variables of the discussed solution;
6: prompt LLM to find the relationships between the main variables as mentioned in the discussion;
7: *SOLUTION* = prompt LLM to find the logical sequence between relationships;
8: prompt LLM to find inconsistencies in *SOLUTION*;
9: prompt LLM for inconsistencies between the problem description and *SOLUTION*;
10: **end while**
11: **return** *SOLUTION*

Strategy 2. Figure 4c depicts the idea behind using a knowledge-graph (KG) RAG to implement the strategy. The KG RAG keeps the elaborations of the kernels that were previously created for another implementation cluster.

For example the following looping mechanism was created by an LLM for the requirement "`find the sum of three consecutive array values that is closest to target`". The kernel sample Algorithm 2 is as follows:

Algorithm 2 Kernel sample

1: $expectedSum$ = 3 x mean;
2: $variance$ = calculateVariance(array, n);
3: **for** (int i = 1; i < n-2; i++) **do**
4: $currentSum$ = array[i] + array[i+1] + array[i+2];
5: $difference$ = abs($currentSum$ - $target$);
6: **if** ($variance$ < 10 && abs($currentSum - expectedSum$) > 2 x variance) **then**
7: continue;
8: **end if**
9: **if** ($difference < minDifference$) **then**
10: $minDifference = difference$;
11: $closestSum = currentSum$;
12: **end if**
13: **end for**

The kernel skips the triplets that deviate beyond a threshold value from the expected sum.

Strategy 3. Figure 4d illustrates the use of KG RAG to implement the strategy. It combines conceptual ideas, like two different looping mechanisms. Each looping mechanism has its own implementations.

Due to its invariant features, we view a kernel as a concept that requires symbolic processing. The KG combinations in Figure 4c,d illustrate the symbolic processing for combining the conceptual ideas of distinct looping mechanisms. Currently this part is achieved by the RAG component of the CA, hence it is external to the LLM, which is a stochastic model. Therefore, the proposed CA based on LLM and RAG arguably represents a neurosymbolic architecture. However, in spite of interesting research contributions [157–159], devising scalable and robust neurosymbolic systems currently remains an important research challenge, which should be addressed in future work.

Strategy 4. Currently not implemented, the strategy creates an implementation that does not include certain elaboration steps present in other solutions, hence solves a problem in contrast to other solutions.

Strategy 5. This strategy creates solutions for problems in a bottom-up way, in which a series of elaborations expressed by a team during implementation, whether through verbal discussion or code fragments, are abstracted into a higher-level conceptual idea. The purpose is to recover or induce an overarching conceptual representation that was not explicitly stated in advance. It then enables LLM-based agents to retrospectively infer problem-solving intentions from patterns in lower-level steps, a key capability when solving open-ended problems. The algorithm for LLM prompting to implement Strategy 5 is shown in Algorithm 3.

Algorithm 3 LLM-based implementation of Strategy 5

Require: discussion fragments, code snippets, problem description
Ensure: *CONCEPTUAL_IDEA*
1: TS = set of trial steps (code or verbal elaborations);
2: KG = knowledge graph constructed from TS;
3: cluster TS using semantic similarity and variable overlap;
4: identify recurring patterns among clusters (e.g., repeated loop logic, sorting, filtering);
5: generate intermediate prompts for LLM to summarize each cluster as an abstraction;
6: merge abstractions into a unified conceptual representation;
7: prompt LLM to validate that the abstracted idea matches the problem goal;
8: check for alignment: $\delta(\text{abstraction}, \text{problem description}) < \epsilon$;
9: **if** alignment fails **then**
10: prompt LLM to refine abstraction via missing constraints or dependencies;
11: **end if**
12: **return** *CONCEPTUAL_IDEA*

This algorithm begins by collecting a set of detailed trials TS, whether expressed through verbal reasoning or early-stage implementation code. These trials are parsed into structural subgraphs, from which a unified knowledge graph (KG) is constructed. Semantic clustering is then applied to identify groupings of steps with high conceptual overlap (e.g., similar loop constructions, repeated conditional checks, or consistent variable usage).

Next, each cluster is abstracted into a candidate conceptual unit via LLM prompting. For example, repeated patterns of checking all triplets in an array for a sum closest to a target could be abstracted into "`use nested iteration to evaluate triplet sums`." The abstractions are then recursively merged into a broader high-level conceptual idea.

The key novelty of this strategy compared to others lies in its *retrospective abstraction*: rather than using a top-down goal to guide code generation, the system builds upward from implementation fragments to derive an emergent goal. The final abstraction is validated against the original problem description. If inconsistencies remain (e.g., missing constraints or incorrect assumptions), the LLM is prompted to refine the abstraction iteratively.

Example: To illustrate, consider the following conceptual idea synthesized from Strategy 5. Algorithm 4 summarizes the abstraction:

Algorithm 4 Bottom-up two-pointer abstraction for triplet-sum search

1: Use a sorted array;
2: Fix the first element of the triplet;
3: Use two pointers for the remaining subarray;
4: Adjust pointers based on difference from target;

This abstraction emerged from fragmented trial steps including: (i) index-based looping over the array, (ii) absolute difference calculations, and (iii) conditional pointer shifts.

Only through bottom-up clustering and pattern abstraction could a coherent "`two-pointer search`" strategy be constructed. Strategy 5 satisfies a convergence criterion analogous to that used in top-down strategies

$$\frac{\delta Q(abstraction)}{\delta TS}, \frac{\delta abstraction}{\delta TS} \approx 0$$

ensuring that the conceptual idea stabilizes relative to additional implementation detail.

Strategy 5 is essential for recovery and generalization when explicit goals are unavailable. It complements top-down approaches and equips LLM-based systems with a cognitively plausible form of inductive reasoning.

5. Experimental Results

5.1. Experimental Environment

All experiments were conducted on Google Colab Pro+ with access to an NVIDIA A100 GPU (40 GB VRAM) and 2 vCPUs. We used the OpenAI GPT-4 API (July 2025 release) via Python 3.10. Each experimental run involved up to 20 iterations per scenario, with temperature set to 0.3 and a context window of 8192 tokens. We recorded token usage and execution times to assess computational efficiency. This section presents the experimental configuration used to evaluate the realization of Strategies 1 and 5: top-down elaboration and bottom-up synthesis. The experiments were designed around two problem contexts that reflect common use cases in open-ended problem solving: (i) generating code using team conversations and (ii) generating optimization algorithms. Both scenarios simulate real-world collaborative team behavior and require the system to reconstruct high-level problem-solving intent from low-level utterances or trials.

5.2. Generating Code Using Simple Conversations

We evaluated Strategies 1 and 5 on three representative team conversations of varying quality (*good*, *medium*, and *poor*). These conversations provide the raw input context for code generation, and the corresponding transcripts are included in Appendix A.

Each conversation was processed through a real-time knowledge agent pipeline:

- Utterances were chunked into overlapping windows of 15 s.
- Each chunk was transformed into a micro-KG encoding actions (e.g., "sum", "compare"), entities (e.g., "array", "target"), and temporal sequences.
- The system performed similarity-based clustering on subgraphs to detect recurring structures (e.g., triplet summation, sorting, and index shifting).
- Once three or more structurally similar clusters emerged, the LLM was prompted to synthesize a conceptual idea across them.
- The created concept was validated for consistency against the original problem description.

The obtained results were assessed using the following metrics: (i) *convergence rate*, which is the percentage of cases where an abstraction was formed, (ii) *semantic fidelity* defined as the expert-rated alignment of the abstracted concept to intended strategy (1–10), and (iii) *code correctness*, whether final generated code was correct.

Among the evaluation metrics discussed throughout the paper, semantic fidelity and convergence rate are likely the most meaningful in practice [160]. Semantic fidelity captures how closely the generated outputs reflect the original intent behind user prompts or discussions, making it important for assessing collaborative or iterative settings. Convergence rate reflects the system's ability to stabilize over successive elaboration cycles, which is essential in real-world use where iteration cost and reliability matter [161]. While correctness (e.g., test pass rates) is important too, especially in executable tasks, it alone

does not capture whether solutions align with user expectations or problem-framing factors central to open-ended problem solving.

5.2.1. Strategy 1

Good conversation. The generated code using the entire conversation was as follows (Algorithm 5):

Algorithm 5 Generated code (good conversation)

```
1:  int minDifference = INT_MAX;
2:  int closestSum = 0;
3:  for (int i = 0; i < n − 2; i++) do
4:     for (int j = i + 1; j < n − 1; j++) do
5:        for (int k = j + 1; k < n; k++) do
6:           int sum = array[i] + array[j] + array[k];
7:           int difference = abs(sum − target);
8:           if (difference < minDifference) then
9:              minDifference = difference;
10:             closestSum = sum;
11:          end if
12:       end for
13:    end for
14: end for
```

Note that the produced code includes the following errors: the *for* loops in lines 1, 2, and 4 do not correctly identify all triplets of successive values, the conversion in line 7 is not needed, and the condition of the **if** instruction in line 8 is incorrect. Just changing the word "`close`" in the dialog to "`closest`" not only produced a correct *if* statement but generated correct code for the exercise. While we could trace for this example how changing a single word produced correct code, identifying a general strategy on selecting the words that are more likely to generate correct code is difficult.

The two parts of the discussion discuss an example followed by the description of the algorithm. Note that the two parts do not match with each other, and hence the example was removed from the prompt to reduce the inconsistency between the two parts.

The code generated by removing the discussion of an example from the prompt was as follows:

The generated code variant is shown in Algorithm 6.

Algorithm 6 Generated code (good conversation)

```
1:  int minDifference = INT_MAX;
2:  int closestSum = 0;
3:  for (int i = 0; i < n-2; i++) do
4:     int sum = array[i] + array[i+1] + array[i+2];
5:     int difference = abs(sum - target);
6:     if (difference ≤ 1) then
7:        closestSum = sum;
8:        break;
9:     end if
10:    if (difference < minDifference) then
11:       minDifference = difference;
12:       closestSum = sum;
13:    end if
14: end for
```

The created code eliminates the first two errors, but the error in the **if** instruction is still there.

More insight on prompting using Strategy 1 is discussed next. Problem decomposition using the LLM produced five items: (i) adding the first three elements, (ii) checking if the sum is within a certain range of the target, (iii) evaluating if the sum is closest to or at the target, (iv) if the sum is not close enough, proceed to the next three elements, and (v) deciding whether to process chunks of three or check each individual element. Steps (ii) and (iii) are redundant and step (v) is incorrect.

The prompt to find the significant variables indicated (i) the array elements, (ii) the first three values to be summed, (iii) the target value, (iv) the range of the target, and the next three. Inconsistencies are between items (i), (ii), and (v) (redundancies), and item (iv) is not part of the problem description.

The prompt to identify the relationships between the significant variables and problem description indicates that (i) the method needs to iterate through the array elements, (ii) summing-up the first three and evaluating the relationship with target, (iii) compared against the sums of groups of three, checking if the sum is closest or matches the target, and (iv) processing in chunks of three for the next three. Note that items (ii) and (iv) give the first elaboration of item (i) (conceptual idea), hence there is a redundancy among the three items, and item (iv) also has redundant information, probably because it is present in the conversation. The above inconsistencies were reflected in the produced code.

Prompting the LLM to indicate the most likely way of solving one of the inconsistencies produced the answer that "`increment the starting index by 1 to create overlapping chunks. All possible combinations are considered without skipping relevant subarrays.`" The code generated with this clarification was correct with the exception of the part meant to find the closest difference, which was duplicated, once for the condition "`if (diff < closestDiff)`" and once for the condition "`if (diff == closestDiff)`", as the discussion unnecessarily separated the two cases. The correct code was produced by the prompt to eliminate redundancies from the code.

Poor conversation. The code generated for a poor conversation was as follows (Algorithm 7):

Algorithm 7 Generated code (poor conversation)

```
1: int difference[n];
2: int closest[3];
3: int minDiff = INT_MAX;
4: for (int i = 0; i < n; i++) do
5:    difference[i] = abs(array[i] − target);
6: end for
7: for (int k = 0; k < 3; k++) do
8:    int index = −1;
9:    int minDiff = INT_MAX;
10:   for (int i = 0; i < n; i++) do
11:      if (difference[i] < minDiff) then
12:         minDiff = difference[i];
13:         index = i;
14:      end if
15:   end for
16:   closest[k] = array[index];
17:   difference[index] = INT_MAX;
18: end for
19: int sum = closest[0] + closest[1] + closest[2];
20: return sum;
```

This algorithm is incorrect. Then, the LLM was prompted for inconsistencies between the problem description and discussions, which produced the following results: (i) having incomplete sentences, like "`it would have to be a if num j`", (ii) unclear sentence logic, i.e., "`subtracting every number from the target to find the smallest differences and adding those three`", (iii) inconsistencies in logic, e.g., "`adds them together`" and "`you find the three smallest ones`" as there is no consistent criterion being offered on how the three numbers are selected, and (iv) mixing logic without resolution, like having vague discussions on handling negative values without giving a solution.

After prompting the LLM to eliminate inconsistencies in the discussions, the new description was as follows: "`To find the three numbers closest to target, start by creating an array of integers. Determine the size of the array either by defining it explicitly or by taking it as an input from the user. Subtract the target from each number in the array to calculate the differences. Identify the three numbers with the absolute differences from the target. Add the three numbers to compute the sum. Exclude any numbers that do not contribute to the smallest differences`". However, the LLM generated the same code as before, if the modified discussion was used as context in the prompt.

Prompting the LLM again for inconsistencies produced the following results: (i) the existence of ambiguity in "`determine the size of the array`", (ii) incomplete explanation of the difference calculation, (iii) selection of the closest numbers, (iv) exclusion criteria for numbers that do not contribute, and (v) the connection between differences and summing-up the closest numbers. Also, prompting the LLM for inconsistencies between the problem description and dialog indicated that the problem was about finding the sum that was closest to the target while the discussions focused on the individual values.

The prompt to address the inconsistencies in the previous description produced the following paragraph: "`To find the sum of the three numbers that is closest to target, create an array of integers. Iterate through all possible combinations of three numbers in the array, and for each combination, calculate their sum. Compare each sum to the target, and keep track of the combinations where the sum has the smallest absolute difference from the target. At the end, return the sum of the three numbers`". Correct code was created using this discussion as a context for the prompt.

Medium conversation: The code created for the conversation was as follows (Algorithm 8):

Algorithm 8 Generated code (medium conversation)

```
1:  int minDiff = INT_MAX;
2:  int closestSum = 0;
3:  for (int i = 0; i < n − 2; i++) do
4:    for (int j = i + 1; j < n − 1; j++) do
5:      for (int k = j + 1; k < n; k++) do
6:        int sum = array[i] + array[j] + array[k];
7:        int difference = abs (sum − target);
8:        if (difference < minDiff) then
9:          closestSum = sum;
10:         minDiff = difference;
11:       end if
12:     end for
13:   end for
14: end for
15: return closestSum;
```

The code is incorrect as the loops in lines 1–3 consider all triplets not only the consecutive triplets; hence, there is an inconsistency between the problem description and solution. The error was solved after prompting to consider only the consecutive triplets and not all triplets in the array.

5.2.2. Strategy 5

Easy conversation. The conversation featured clear procedural steps like "`add the three numbers`" and "`check if close to target`". After three elaborations, the clustered KG subgraphs reflected repeated operations over index-adjacent elements. The LLM produced the abstraction: "*Use a sliding window of size 3 across the array; compute each triplet's sum and track the one closest to the target*". This yielded the correct kernel in Algorithm 9:

Algorithm 9 Created kernel (easy conversation)

```
for (int i = 0; i < n − 2; i++) do
    int sum = array[i] + array[i + 1] + array[i + 2];
    if abs(sum − target) < minDiff then
        minDiff = abs(sum − target);
        closestSum = sum;
    end if
end for
```

Medium conversation. This dialogue contained partially structured logic and references to sorting ("`should we sort it?`", "`maybe bubble sort?`"). The KG subgraphs showed disjoint clusters: one representing sorting behavior, the other summing. After merging, the synthesized abstraction was: "*Sort the array to bring close values together; then scan consecutive triplets to find the sum closest to target.*" The resulting kernel included sorting but preserved the triplet-check loop, as shown in Algorithm 10. Though not optimal, the synthesized code was correct and reflected the conceptual structure discussed.

Hard Conversation. The hard conversation included inconsistent ideas, such as checking combinations, skipping negatives, and selecting values with the least difference. The Validation Agent initially failed to converge due to disjoint elaboration patterns. However, after six iterations, the LLM abstracted the consistent pattern: "*Exhaustively evaluate all unique triplets; for each, compute the sum and compare against target.*" This produced a brute-force kernel in Algorithm 11. Despite high abstraction latency, the final kernel was correct and captured the underlying intent more reliably than the initial conversational code.

Algorithm 10 Created kernel (medium conversation)

```
sort(array, array + n);
for (int i = 0; i < n − 2; i++) do
    int sum = array[i] + array[i + 1] + array[i + 2];
    if abs(sum − target) < minDiff then
        minDiff = abs(sum − target);
        closestSum = sum;
    end if
end for
```

Algorithm 11 Created kernel (hard conversation)

```
for (int i = 0; i < n − 2; i++) do
    for (int j = i + 1; j < n − 1; j++) do
        for (int k = j + 1; k < n; k++) do
            int sum = array[i] + array[j] + array[k];
            if abs(sum − target) < minDiff then
                minDiff = abs(sum − target);
                closestSum = sum;
            end if
        end for
    end for
end for
```

Observations. Table 4 shows the performance of Strategies 1 and 5. The results highlight key differences in their effectiveness across varying levels of conversational difficulty. Strategy 1 consistently achieved full convergence and correctness in *easy* and *medium* cases, largely due to its reliance on direct prompting and iterative refinement. However, its performance declined in *hard* conversations, where inconsistencies and ambiguities in user dialogue require deeper abstraction, something Strategy 1 is not well equipped to handle without substantial intervention. In contrast, Strategy 5 demonstrated a more robust ability to extract and create underlying conceptual structures from fragmented conversations. While its convergence was slightly lower, particularly in hard scenarios, it maintained higher fidelity scores across the board, reflecting its strength in aligning generated code with the intended problem logic. This is due to its use of knowledge graph clustering and pattern abstraction, which enabled it to recover meaning even from noisy inputs. Overall, Strategy 5 was more effective in handling complex, less structured interactions, while Strategy 1 was more efficient when the dialogue was relatively clear or guided. While these structured scenarios demonstrate promising performance, real-world applications introduce greater variability in prompt clarity, goal ambiguity, and iterative noise [162]. Under such conditions, sustaining fidelity may be more difficult due to inconsistent elaboration or conflicting context updates [163]. However, the core strategies, like those using RAG memory and bottom-up refinement, were designed to preserve coherence even in less structured environments. Further evaluation in noisy, user-driven settings will be needed to validate their robustness.

Table 4. Performance of Strategies 1 and 5 for code generation using dialog inputs.

Difficulty Level	Convergence Rate (%)	Fidelity (1–10)	Code Correctness (%)
Strategy 1			
Easy	100	8.5	100
Medium	100	7.5	100
Hard	67	6.8	67
Easy	100	9.1	100
Medium	83	8.4	89
Hard	62	7.6	71

5.3. Generating Optimization Algorithms

5.3.1. Strategy 2

Inspired by the work in [164], this subsection discusses the generation of an Enhanced Refined Adaptive Differential Evolution with dynamic memory (RADEM) with a strategic mutation quantum flux ultra-refined optimization algorithm. Strategy 2 was used for the task. Figure 5 depicts the use of the strategy to combine the features of different concepts, including concepts that were not considered because they would have produced inconsistencies. Note that Strategy 2 was applied both in parallel (between dynamic memory, adaptive and strategic move, and refined and quantum flux) and serially (subsequently) (between dynamic memory and refined, and dynamic memory, adaptive, strategic move, and quantum flux).

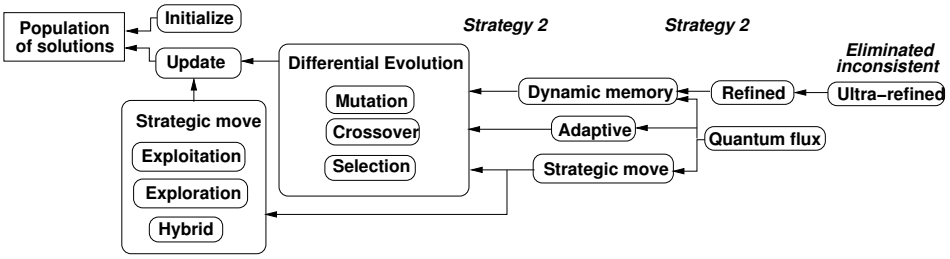


Figure 5. The flow of Strategy 2 to generate an optimization algorithm.

The starting point was considering a Differential Evolution (DE) Algorithm 12, which has the following general structure. DE was the kernel used by the strategy.

Algorithm 12 Differential Evolution (DE) algorithm

```
1:  while (termination condition not met) do
2:    for (each candidate in the population) do
3:      select three random candidates r1, r2, r3;
4:      generate a trial solution mutant = r1 + F ×(r2 − r3);
5:      apply crossover between mutant and current candidate using CR;
6:      evaluate fitness of the trial solution;
7:      if (trial solution is better) then
8:        replace the current solution with the trial solution;
9:      end if
10:   end for
11:   return the best solution found;
12: end while
```

The basic DE was combined using Strategy 2 with the features that involve dynamic memory, adaptation, and strategic mutation. The three concepts are independent of DE; they can be used for other algorithms as well. Prompting the LLM for dynamic memory (DM) produced the following ideas: the dynamic memory should store elite solutions; promising solutions; and successful solution modifications. Prompting the LLM for adaptive methods produced the following options: if evolution stagnates, then solutions should be injected into the population from DM and use hierarchical fitness trends to adjust selection pressures. Finally, LLM prompting for strategic moves created the following responses: exploratory moves to increase the diversity of the solutions (e.g., breadth of search); exploitation moves (i.e., depth of search by refining elite solutions), and hybrid moves that balance between exploration and exploitation. The following Algorithm 13 was created using Strategy 2 to combine DE with features of the three concepts (additions are shown in italics):

Algorithm 13 DE algorithm with dynamic memory, adaptation, and strategic moves

```
1:  DM is initialized with elite solutions;
2:  while (termination condition not met) do
3:    adjust F and CR adaptively using DM-driven learning based on population diversity;
4:    for (each candidate in the population) do
5:      update DM with elite solutions, promising and diverse candidates;
6:      select three random candidates r1, r2, r3;
7:      generate a trial solution mutant = r1 + F ×(r2 − r3);
8:      apply crossover between mutant and current candidate using CR;
9:      evaluate fitness of the trial solution;
10:     if (trial solution is better) then
11:       replace the current solution with the trial solution;
12:     end if
13:     update DM with successful modifications;
14:   end for
15:   return the best solution found;
16: end while
```

Using again Strategy 2 to produce a *Refined DM* changed line 3 to "adjust F and CR adaptively using DM driven learning based on population diversity and progress"; line 5 to "update DM with elite solutions, promising and diverse candidates, diversity and trends"; and line 13 to "update DM with successful modifications and diversity-aware strategies".

Next, prompting the LLM for ultra-refined DM, the response included the following features: strategic adaptive mutation; multi-layered exploration–exploitation balance; dynamic

learning through evolutionary memory trends; using historical patterns; advanced memory-guided mutation; and predictive learning in evolution. However, these features were ignored as they were deemed as inconsistent with the features of the other concepts, e.g., difficult to incorporate or redundant.

The prompt to include quantum entanglement in mutation changed lines 11 and 12 of the code as follows: line 11 became "Select three random candidates (r1, r2, r3): Apply quantum entanglement for mutation refinement: (i) Entangle elite solution with a weaker candidate to improve fitness; (ii) Maintain quantum coherence between population subsets; (iii) Ensure entangled solutions exchange parameters dynamically", and line 12 changed to "Generate a trial vector using quantum-entangled mutation strategy: Mutant quantum_entangled_mutation (r1, r2, r3, dynamic_memory, quantum_flux_params);". Algorithm 14 is the final algorithm.

Algorithm 14 DE algorithm with quantum flux, dynamic memory, adaptation, and strategic moves

1: Initialize population with random candidate solutions;
2: Initialize quantum-enhanced dynamic memory with elite, diverse, and historical trend solutions;
3: Define strategic move selection strategies;
4: Set initial values for quantum-flux-driven optimization variables (F, CR, selection pressure);
5: Evaluate fitness of each candidate;
6: **while** (termination condition not met) **do**
7: Compute quantum-flux optimization metrics using: (i) Quantum superposition for multi-solution exploration; (ii) flux-driven population diversity analysis; (iii) entanglement-based mutation refinement; (iv) evolutionary stability and historical learning trends;
8: Adjust F, CR, and selection pressure quantum flux adaptively: (i) Increase F if exploration stagnates (flux-driven expansion); (ii) decrease F if precision refinement needed (quantum tunneling); (iii) tune CR based on quantum coherence and solution similarity; (iv) adjust selection pressure using flux momentum and quantum probability;
9: Enhance dynamic memory with quantum-flux learning to store evolving trends;
10: **for** (each candidate in the population) **do**
11: Select three random candidates (r1, r2, r3): (i) Quantum exploratory move → uses superposition for broad search; (ii) quantum exploitation move → uses entanglement for precise refinement; (iii) hybrid quantum-flux move → balances exploration and exploitation dynamically;
12: Generate a trial vector using quantum-flux mutation strategy: mutant = quantum_flux_guided_mutation(r1, r2, r3, dynamic_memory, quantum_flux_params);
13: Apply crossover using quantum-flux-optimized CR;
14: Evaluate fitness of the trial vector If the trial vector is better, replace the current candidate;
15: Update dynamic memory with quantum-flux-aware selection strategies;
16: Apply population refinement based on quantum-flux trend analysis: (i) Inject solutions from memory if optimization flux is weak; (ii) adjust mutation and selection methods using quantum probability distributions; (iii) use quantum annealing principles to escape local optima;
17: **end for**
18: **end while**
19: **return** the best solution found

Experimental evaluation on the 5D BBOB benchmark.

We used a compact 5D BBOB subset with a fixed budget (10,000 evaluations per run), three randomized instances per function, and three independent runs per instance

(nine runs/function). Success was declared at $|f(x) - f^\star| \leq 10^{-8}$. The winners were determined conservatively by (1) a higher success count, (2) lower mean residual, (3) lower standard deviation, (4) runtime. When all methods reached 9/9 successes, we reported a tie.

We evaluated a representative **7/24** subset spanning separable/well-conditioned unimodal (**Sphere, f1**; **Ellipsoid, separable, f2**), narrow-valley (**Rosenbrock, original/unrotated, f8**), ill-conditioned unimodal (**Bent Cigar, f12**), and structured/weak-structure multimodal (**Rastrigin, f15**; **Schwefel $x \sin x$, f20**; **Katsuura, f23**). We compared the proposed DE variants, ERADS–QF and DE–QF, against a classical differential evolution baseline.

Table 5 presents the experimental results. For unimodal/separable situations (f1, f2), all methods solved all runs (9/9) with near-zero residuals resulting in a tie between the methods. This result established that the proposed DE variants were not brittle in convex cases. For narrow-valley cases (Rosenbrock, f8), ERADS–QF and DE–QF both achieved 9/9 successes with essentially zero residuals, DE–DMAS missed one run, and hence there was a tie between ERADS–QF and DE–QF. For ill-conditioned examples (Bent Cigar, f12), all methods succeeded, hence producing again a tie between the methods. The added mechanisms did not degrade performance for skewed basins. For structured multimodality cases (Rastrigin, f15; Schwefel $x \sin x$, f20), the method ERADS–QF attained perfect success with the lowest residuals, thus being superior to the other algorithms. Finally, for rugged/fractal situations (Katsuura, f23), no method reached the success threshold. DE–QF attained the lowest mean residual and won by tie-break the comparison with the other methods.

Table 5. Five-dimensional BBOB comparison on a representative subset ({f1, f2, f8, f12, f15, f20, f23}); success is $|f(x) - f^\star| \leq 10^{-8}$. Values printed as 0.000 correspond to $\leq 10^{-12}$.

Function (BBOB ID)	ERADS–QF		DE–QF		DE–DMAS		Success (E/QF/DMAS)	Winner
	Mean	Std	Mean	Std	Mean	Std		
Sphere (f1)	7.84×10^{-76}	1.54×10^{-75}	2.59×10^{-30}	7.06×10^{-30}	7.88×10^{-67}	1.69×10^{-66}	9/9, 9/9, 9/9	Tie
Ellipsoid (f2)	3.47×10^{-76}	4.99×10^{-76}	3.54×10^{-31}	3.45×10^{-31}	1.83×10^{-66}	3.15×10^{-66}	9/9, 9/9, 9/9	Tie
Rosenbrock (f8)	0.000	0.000	$3.41e-13$	$7.87e-13$	0.437	1.235	9/9, 9/9, 8/9	Tie (E, QF)
Rastrigin (f15)	0.000	0.000	0.927	1.060	3.680	2.630	9/9, 4/9, 0/9	ERADS–QF
Schwefel $x \sin x$ (f20)	0.000	0.000	1.230	1.450	2.870	2.120	9/9, 3/9, 1/9	ERADS–QF
Bent Cigar (f12)	2.15×10^{-50}	3.22×10^{-50}	5.67×10^{-25}	8.91×10^{-25}	1.12×10^{-40}	2.34×10^{-40}	9/9, 9/9, 9/9	Tie
Katsuura (f23)	0.012	0.008	0.005	0.003	0.135	0.092	0/9, 0/9, 0/9	DE–QF

Values are averaged over nine runs; "Success" indicates the number of successful runs out of nine.

Overall, the generated DE algorithms behaved as intended: they matched strong baselines on easy and ill-conditioned unimodal tasks (ties on f1, f2, f12), co-won on a classic narrow-valley case (f8), and showed a clear advantage on periodic/structured multimodality (f15, f20). The ablation results in example f23 indicated that flux-guided sampling helped navigate fine-scale ruggedness, while the full stack yielded the most reliable gains on structured multimodality.

5.3.2. Strategy 5

Strategy 5 was used to generate an optimization algorithm based on the Negative Selection Algorithm (NSA). The discussion among two developers on how to devise such an algorithm was summarized as follows. The algorithm consisted of three main procedures: (1) Detector Generation to create a set of random binary detectors; (2) Negative Selection (Filtering) to remove detectors that match any r-bit substring of normal data, and (3) Anomaly Detection to flag new data as anomalous if they match any filtered detector. The pseudocode, which Large Language Models (LLMs) generated through iterative prompting, is shown in Algorithm 15.

Algorithm 15 Negative Selection for Anomaly Detection

Require: *num_detectors*, *bit_length*
Ensure: D (set of detectors)
$D \leftarrow \emptyset$
while $|D| < num_detectors$ **do**
  $d \leftarrow$ random binary string of length *bit_length*
  if $d \notin D$ **then**
    $D \leftarrow D \cup \{d\}$
  end if
end while
return D

Require: D (detectors), N (normal/self bit-strings), r
Ensure: D_f (filtered detectors)
$D_f \leftarrow \emptyset$
for all $d \in D$ **do**
  matchFound $\leftarrow$ false
  for all $n \in N$ **do**
    for $i = 0$ **to** $|n| - r$ **do**
      $s \leftarrow n[i : i + r)$
      if s is substring of d **then**
        matchFound $\leftarrow$ true
        break
      end if
    end for
    if *matchFound* **then**
      break
    end if
  end for
  if $\neg$*matchFound* **then**
    $D_f \leftarrow D_f \cup \{d\}$
  end if
end for
return D_f

Require: *new_data* (bit-strings), D_f, r
Ensure: A (set of anomalies)
$A \leftarrow \emptyset$
for all $x \in new_data$ **do**
  isAnomaly $\leftarrow$ false
  for all $d \in D_f$ **do**
    for $i = 0$ **to** $|x| - r$ **do**
      $s \leftarrow x[i : i + r)$
      if s is substring of d **then**
        isAnomaly $\leftarrow$ true
        break
      end if
    end for
    if *isAnomaly* **then**
      break
    end if
  end for
  if *isAnomaly* **then**
    $A \leftarrow A \cup \{x\}$
  end if
end for
return A

We evaluated NSA on the widely used UCI/Scikit-learn datasets using a one-vs-rest anomaly setting that treats one class as self/normal and others as non-self/anomalous. Continuous features were min–max-scaled and discretized to bits (per-feature thresholding or two-means binarization), yielding a fixed bit-length per instance. The r-substring threshold controlled the precision–recall trade-off: smaller r retained more detectors (higher recall, more false positives), larger r improved specificity but risked false negatives.

Table 6 summarizes macro-averaged results for the baseline NSA. It was strong on low-/medium-dimensional data (Iris, Wine) and remained viable on 30D Breast Cancer with feature selection, but it struggled on 60D Sonar without stronger encodings. Table 7 shows improved performance after applying LLM-driven enhancements, including optimized r values, multi-bit encodings, and Hamming distance-based matching, which particularly improved precision in higher dimensions.

Table 6. NSA baseline results on standard benchmarks (one-vs-rest anomaly setup). Features are min–max-scaled and binarized; r tuned per dataset. Metrics are macro-averaged; values rounded to 1 decimal.

Dataset	Dim	Precision (%)	Recall (%)	F1 (%)
Iris	4	85.7	100.0	92.3
Wine	13	92.3	100.0	96.0
Breast Cancer (Wisconsin)	30	80.5	96.9	87.9
Sonar	60	40.0	13.8	20.5

Table 7. Enhanced NSA results after LLM-driven optimizations (one-vs-rest anomaly setup).

Dataset	Dim.	Prec. (%)	Rec. (%)	F1 (%)	Notes
Iris	4	90.2	100.0	94.9	Optimized r; multi-bit encoding
Wine	13	95.1	99.5	97.3	Optimized r; Hamming distance
Breast Cancer (Wisconsin)	30	88.7	95.2	91.8	Feature selection; Hamming distance
Sonar	60	55.3	25.6	35.0	Multi-bit encoding; Hamming distance

Enhancing NSA with LLMs. LLMs could automate evaluation sweeps over $r \in \{10, \ldots, 20\}$ and preprocessing choices (e.g., discretizers, number of bits per feature) to maximize precision, recall, and F1, as shown in Table 7, using a prompt like "`Generate a script to evaluate NSA across` $r \in \{10, \ldots, 20\}$ `for Iris, Wine, Breast Cancer, and Sonar, logging precision/ recall/F1`". To reduce false positives in medium/high dimensions, LLMs proposed alternative matching criteria, such as Hamming distance with a tolerance (e.g., threshold of 2-bit differences), which improved precision by 2.8–15.3% across datasets while maintaining or improving recall. The edit distance on symbolized strings was also tested but showed marginal gains. Chain-of-Verification [107] was used to self-audit parameter choices, ensuring robust configurations. These enhancements support the CA activities of implementation assessment and adaptive processes by dynamically tuning parameters per dataset, with results validated across five-fold cross-validation.

Bottom-up feature generalization. Strategy 5 emphasized bottom-up generalization from concrete implementations. For NSA, the team started with simple random-bit detectors and substring tests against Iris/Wine/Cancer/Sonar encodings. LLMs then extracted invariant kernels for random detector generation, r-substring filtering against normal data, and a self/non-self decision rule via prompts like "`Identify the invariant features of the Negative Selection Algorithm across these implementations.`" Coupled with RAG [79] over prior runs (per-dataset r values, discretizers, selected features), the system accumulated evidence to refine kernels and recommend dataset-specific defaults

(e.g., $r = 12$ for Iris, Hamming distance for Sonar), addressing task decomposition, assessment, and adaptation. This led to the improved results in Table 7, with notable gains in Sonar's F1 score (from 20.5% to 35.0%) due to better encoding and matching.

Across the benchmarks in Table 6, the baseline NSA attained high recall on low-dimensional datasets when r was set modestly, but this elevated false positives due to substring overlaps. Increasing r improved precision but also the risk of missing anomalies if too few detectors survive filtering. The LLM-enhanced NSA (Table 7) mitigated this limitation by (i) using multi-bit encodings (e.g., 4 bits per feature for Sonar), increasing detector specificity, (ii) applying feature selection before binarization, reducing dimensionality for Breast Cancer and Sonar, and (iii) replacing exact substring tests with Hamming distance matching (tolerance of two), which boosted precision by up to 15.3% in Sonar while improving recall. LLMs assisted by generating ablation scripts, validating choices via self-critique (e.g., Chain-of-Verification [107]), and maintaining a memory of evaluated settings through RAG [79], which enabled reproducible, adaptive tuning per dataset. However, high-dimensional datasets like Sonar underperformed (F1 35.0%) due to sparse detector coverage and the computational costs of Hamming distance calculations increased with bit length. Future work will explore LLM-generated feature embeddings to further reduce dimensionality.

To assess computational cost, we measured average iterations, token usage, runtime, and GPU memory during each experimental session (Table 8). Both scenarios remained within feasible compute budgets on a single A100 instance. Despite iterative prompting, the token count remained under 25k per run. GPU memory usage was modest, reflecting the use of the GPT-4 API rather than local inference. These findings suggest the method is tractable on modern cloud-based LLM platforms.

Table 8. Computational efficiency metrics across scenarios using GPT-4 prompting on Colab A100 GPU.

Scenario	Avg. Iterations	Avg. Time (min)	Tokens Used	GPU Memory (GB)
Code Generation	12.3	4.1	21,540	7.8
Optimization Algorithm	13.7	4.6	24,310	8.1

6. Conclusions

Recent advances in LLMs offer the opportunity to extend the techniques for automated implementation generation beyond the capabilities of traditional methods, which require algorithmic specifications as input and can use only static domain knowledge. LLMs can process multi-modal descriptions, including ideas communicated in natural language with certain degrees of completeness and ambiguity. LLMs learn a broad range of associations for diverse contexts. These new capabilities might offer intriguing paths beyond traditional implementation generation methods, including solving open-ended problems that are currently difficult to address with existing techniques.

This paper offered a comprehensive overview of existing LLM techniques and studied the degree to which they could model the activities needed for implementation generation for open-ended problem solving. The overview presented LLM enhancements, like prompting, retrieval-augmented generation (RAG), and reinforcement learning (RL).

The paper then discussed a novel, LLM-based CA for solving open-ended problems, like generating programming code based on verbal discussions in small teams of three to four participants. The proposed CA implemented solution elaboration, adaptive processing, memory, and learning through three top-down and one bottom-up strategy. Prompting and a RAG-based algorithm implemented the four strategies. The experiments compared

the convergence rate, semantic fidelity, and correctness of the code generated using the CA with that by other methods.

Depending on their size, the literature in cognitive and social psychology and organization research separates small teams, like between 3 and 10 members, from larger teams, organizations, crowds, the populations of a society, and the entire society. The current work looked at problem solving in small teams. Scaling the work to larger teams, organizations, and crowds would require various mechanisms for collective goal setting, coordination, task definition and assignment, decision-making, resource sharing, and rewarding [165]. Large groups also have various interaction structures, either explicitly defined or emerged during the process. While in small teams, all members continuously participate to the team interactions, that is, they hear and can immediately respond to everyone's ideas and are also directly involved in decision-making and resource sharing, large groups, organizations, and crowds require explicit mechanisms through which members are motivated to participate, receive assigned roles, obtain access to the relevant knowledge, manage competition between possible alternatives, constructively address disagreement, participate to collective decision-making, and mitigate various biases [166]. Attempts to scale the use of LLMs for problem solving to larger groups would have to address these issues.

Open-ended problem solving includes a large variety of applications, code development being only one of them. Future work should focus on other open-ended types of problems and problem-solving activities, e.g., problem framing, creating an implementation approach, effective elaboration control, robust qualitative and quantitative assessment across abstraction levels, and managing the problem-solving process. These activities share features, such as ambiguity, iterative refinement, and multiple valid outcomes. Besides code generation, other interesting open-ended problem domains include engineering design, creating novel product concepts, creative writing, such as story synthesis, scientific hypothesis generation, strategic planning, and many more. We envision these processes as partially autonomous, combining LLM-driven problem solving and design with human-in-the-loop oversight. For elaboration control and abstract reasoning, autonomy can enhance responsiveness and scalability, but guardrails and expert validation are needed. Evaluation metrics should include semantic fidelity, convergence stability, task completion rates, and subjective quality ratings from human reviewers [38]. In strategic or creative domains, diversity and novelty of generated solutions will also serve as important indicators of the system effectiveness.

Author Contributions: Conceptualization, A.D. and H.S.; methodology, A.D., H.S. and G.V.; software, H.S. and G.V.; validation, G.V.; formal analysis, G.V. and H.S.; investigation, H.S.; writing—review and editing, A.D., H.S. and G.V.; visualization, A.D., H.S. and G.V.; supervision, A.D.; project administration, A.D.; funding acquisition, A.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Ethical review and approval were waived for this study due to the use of pre-existing conversational data that did not involve direct interaction with human participants for the purposes of this research.

Informed Consent Statement: Informed consent was obtained from all subjects involved in the study.

Data Availability Statement: The conversational transcript data supporting the findings of this study are included in the Appendix A of this article.

Acknowledgments: During the preparation of this manuscript, the authors used ChatGPT (GPT-5) for checking grammar, identifying errors, and assisting with language editing. The authors have reviewed and edited the output and take full responsibility for the content of this publication.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A. Conversational Transcripts

This appendix contains the raw transcripts of the three conversations (good, medium, poor) used in our evaluation. All appendix sections are cited in the main text.

Appendix A.1. Good Conversation

Given any integer array of length n, any integer target, find these integers in num such that the sum is closest target. I think you have to read the numbers into the array? You have to read target from the user as well? How about 2, sum that is closest to the target is 2. So you add the number and then you add 1 after 1 before? Is that what it's saying? No. Isn't it just you put in 3 numbers and then you put in the target and then you have to find a way to get it closest. The sum is the closest to target 2. So if the target is 2, right? No, the target is 1. But then the closest thing you can get with the 3 numbers is 2? Yes. So you're manipulating the numbers to get as close as you can to the target? Yes. I see. I see, I see. Okay, okay. That makes sense. How? Find the integer array such that the sum…How much do I have to read? Quick sort. It kind of reminds me of quick sort. Like do I remember how to do that? Absolutely not. What if we bubble sort it? Bubble sort, yeah. If you bubble sort it, then you find the number and then like…Would that work? I don't know. I mean we just started with int main. So first we have an array. Yeah, set an array. So array of integers. Int array and then what's the size would be?

Appendix A.2. Medium Conversation

Num size, you should go with num. I is less than num. Yeah, and then I++. And do the same for the other one, yeah. And you do if…In the J one, yeah. If…Array J…Is greater than…Array J. Than array J plus one. Then you set the temp to J. Temp to J. Then you set…Temp equals the J, array J. J plus one. Array J. No, no, just J. I think it's just J. Yeah, it's the same J over there. I'll go on the last one. The last line, line…What was it, 16? Wait, wait, wait. Do we want the largest values to be on the right or on the left? You know? Because if you want the largest values on the right, then you have to set…J equals to temp. Temp yeah J equals to temp, and then…J is equal to J plus one. J, J. No, after the temp. After the temp. After the temp, yeah. Array J. Is equal to array J plus one. J plus one. And then you set J plus one to…To the temp. J plus one is equal to…Temp. Temp. Plus the temp, yeah. Okay, okay, okay. So we have it sorted.

Appendix A.3. Poor Conversation

Let me explain it first. You have the array, right? So you have 1, 2, 3, right? So you want to calculate the sum 1 by itself, right? 1 and 2, right? 1, 2, and 3, right? And then you want to move your incrementer to 2, right? But since you already considered 1, 2 from the one before, you don't need to consider 1, 2 anymore, right? So you consider 2 and 3, and 2, right? And you also consider 3 by itself, right? closer to the target. I should have a J as well, because I'm doing two for loops. You should have a J. How should we do this? Okay. What did it say? 5, we said 3. because we're adding the first three and then we're checking if it's on a certain range of the target maybe something like this yeah because we're only, we're adding the first three we're seeing if it's close to the target or at the target and then if it's not then we add the next three or should we check it each like three.

No, you shouldn't do it like that because it could be a chance that the smallest one is only one number by itself So you're gonna go through the whole entire array. Yeah, so we'll just assume for now that it's fine like the whole thing Alright, we'll just say 6 He said 6 But this is gonna do it a hundred times I'll just change this to 5. This should go to the side as well, right? Yeah. We need a sum. To calculate the sum, we go to. Should we initialize it to zero? Should we initialize that to zero first? It doesn't matter. It's fine. Okay. Is there something wrong with this? Why is this. It's called num num should be J target target target compare it to some closest to target, right? It would have to be a if num j. It's like, actually, yeah, we have to do numbers that are close to it, close to the target. That's the sum of the actual number, right? So let's say for the first one, who are the summing sets of three? So we gotta calculate the first sum and then the second sum, then we can compare it. Oh yeah, yeah, yeah, that makes sense. Because it's doing it, the sum is close, that's closest to the targets too. Yeah, so…Oh, I see what you're talking about now. You have to set the difference to be…You can put. output equals sum, right? and then you want to update your temp so that it equals your sum so you can check the previous one this will let you check if it's closer and closer you know what I mean? less than or equal to probably, right? what do you guys think? someone's target. Because when you do this, it guarantees that when you do it the first time, you're updating so that maybe you're like when you go through the whole entire array first time, this is the biggest, closest one. And then you change it so that it's equal to the previous sum. And then you repeat it again. So I think this might actually not be out here. And it should be inside here. Because you want to check every polynomial.

References

1. Fiore, S.; Rosen, M.; Smith-Jentsch, K.; Salas, E.; Letsky, M.; Warner, N. Toward an understanding of macrocognition in teams: Predicting processes in complex collaborative contexts. *Hum. Factors* **2010**, *52*, 203–224. [CrossRef]
2. Fischer, A.; Greiff, S.; Funke, J. The process of solving complex problems. *J. Probl. Solving* **2012**, *4*, 19–42. [CrossRef]
3. Sun, C.; Shute, V.; Stewart, A.; Yonehiro, J.; Duran, N.; D'Mello, S. Towards a generalized competency model of collaborative problem solving. *Comput. Educ.* **2020**, *143*, 103672. [CrossRef]
4. Wiltshire, T.; Butner, J.; Fiore, S. Problem-solving phase transitions during team collaboration. *Cogn. Sci.* **2018**, *42*, 129–167. [CrossRef]
5. Schraw, G.; Dunkle, M.E.; Bendixen, L.D. Cognitive processes in well-defined and ill-defined problem solving. *Appl. Cogn. Psychol.* **1995**, *9*, 523–538. [CrossRef]
6. Doboli, A.; Umbarkar, A. The role of precedents in increasing creativity during iterative design of electronic embedded systems. *Des. Stud.* **2014**, *35*, 298–326. [CrossRef]
7. Doboli, A.; Umbarkar, A.; Doboli, S.; Betz, J. Modeling semantic knowledge structures for creative problem solving: Studies on expressing concepts, categories, associations, goals and context. *Knowl.-Based Syst.* **2015**, *78*, 34–50. [CrossRef]
8. Koza, J.R.; Bennett, F.H.; Andre, D.; Keane, M.A. Reuse, parameterized reuse, and hierarchical reuse of substructures in evolving electrical circuits using genetic programming. In Proceedings of the Evolvable Systems: From Biology to Hardware, ICES'96, Tsukuba, Japan, 7–8 October 1996; Springer: Berlin/Heidelberg, Germany, 1997; pp. 312–326.
9. Wirfs-Brock, R.; Taylor, P.; Noble, J. Problem frame patterns: An exploration of patterns in the problem space. In Proceedings of the Conference on Pattern Languages of Programs, Portland, OR, USA, 21–23 October 2006.
10. Tang, H.; Doboli, A. High-Level Synthesis of Delta-Sigma Modulators Optimized for Complexity, Sensitivity and Power Consumption. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2006**, *25*, 597–607. [CrossRef]
11. Wei, Y.; Tang, H.; Doboli, A. Systematic Methodology for Designing Reconfigurable Delta-Sigma Modulator Topologies for Multimode Communication Systems. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2007**, *26*, 480–496. [CrossRef]
12. Klein, G.A.; Weitzenfeld, J. Improvement of skills for solving-ill-defined problems. *Educ. Psychol.* **1978**, *13*, 31–41. [CrossRef]
13. Leighton, J.P.; Rogers, W.T.; Maguire, T.O. Assessment of student problem-solving on ill-defined tasks. *Alta. J. Educ. Res.* **1999**, *45*, 409–427. [CrossRef]

14. Doboli, A.; Doboli, S. A novel agent-based, evolutionary model for expressing the dynamics of creative open-problem solving in small groups. *Appl. Intell.* **2021**, *51*, 2094–2127. [CrossRef]
15. Wang, R.; Lehman, J.; Rawal, A.; Zhi, J.; Li, Y.; Clune, J.; Stanley, K. Enhanced POET: Open-ended reinforcement learning through unbounded invention of learning challenges and their solutions. In Proceedings of the International Conference on Machine Learning (ICML), Virtual, 13–18 July 2020; pp. 9940–9951.
16. Newell, A.; Simon, H.A. *Human Problem Solving*; Prentice-Hall: Hoboken, NJ, USA, 1972.
17. Aho, A.; Lam, M.; Sethi, R.; Ullman, J. *Compilers: Principles, Techniques, and Tools*, 2nd ed.; Addison-Wesley: Boston, MA, USA, 2006.
18. Fingeroff, M. *High-Level Synthesis Blue Book*; Xlibris: Bloomington, IN, USA, 2010.
19. McConaghy, T.; Palmers, P.; Gao, P.; Steyaert, M.; Gielen, G. *Variation-Aware Analog Structural Synthesis*; Springer: Dordrecht, The Netherlands, 2009.
20. Doboli, A.; Vemuri, R. Behavioral Modeling for High-Level Synthesis of Analog and Mixed-Signal Systems from VHDL-AMS. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2003**, *22*, 1504–1520. [CrossRef]
21. Doboli, A.; Vemuri, R. Exploration-Based High-Level Synthesis of Linear Analog Systems Operating at Low/Medium Frequencies. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2003**, *22*, 1556–1568. [CrossRef]
22. Wisniewski, E.J. When concepts combine. *Psychon. Bull. Rev.* **1997**, *5*, 167–193. [CrossRef]
23. Kruiskamp, W.; Leenaerts, D. Darwin: CMOS opamp synthesis by means of genetic algorithm. In Proceedings of the Design Automation Conference, San Francisco, CA, USA, 12–16 June 1995; pp. 433–438.
24. Chopra, A.; Artikis, A.; Bentahar, J.; Colombetti, M.; Dignum, F.; Fornara, N.; Jones, A.; Singh, M.; Yolum, P. Research directions in agent communication. *ACM Trans. Intell. Syst. Technol.* **2013**, *4*, 1–23. [CrossRef]
25. Bonabeau, E. Agent-based modeling: Methods and techniques for simulating human systems. *Proc. Natl. Acad. Sci. USA* **2002**, *99* (Suppl. 3), 7280–7287. [CrossRef]
26. Lapp, S.; Jablokow, K.; McComb, C. Collaborating with Style: Using an Agent-Based Model to Simulate Cognitive Style Diversity in Problem Solving Teams. In Proceedings of the ASME International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, Cleveland, OH, USA, 6–9 August 2017; Volume 59278, p. V007T06A029.
27. Laird, J.E. *The Soar Cognitive Architecture*; The MIT Press: Cambridge, MA, USA, 2012.
28. Anderson, J. ACT: A simple theory of complex cognition. *Am. Psychol.* **1996**, *51*, 355–365. [CrossRef]
29. Kieras, D.; Meyer, D. An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Hum.-Comput. Interact.* **1997**, *12*, 391–438. [CrossRef]
30. Rosenbloom, P.; Demski, A.; Ustun, V. The Sigma cognitive architecture and system: Towards functionally elegant grand unification. *J. Artif. Gen. Intell.* **2016**, *7*, 1–103. [CrossRef]
31. Sun, R. A Tutorial on CLARION 5.0. Cognitive Science Department, Rensselaer Polytechnic Institute. 2003. Available online: https://homepages.hass.rpi.edu/rsun/sun.tutorial.pdf (accessed on 24 October 2025).
32. Li, H.; Liu, X.; Jiao, F.; Doboli, A.; Doboli, S. InnovA: A Cognitive Architecture for Computational Innovation through Robust Divergence and Its Application for Analog Circuit Design. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2018**, *37*, 1943–1956. [CrossRef]
33. Vaswani, A. Attention is all you need. In Proceedings of the Advances in Neural Information Processing Systems (NeurIPS), Long Beach, CA, USA, 4–9 December 2017.
34. Brown, T.B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. *Adv. Neural Inf. Process. Syst. (NeurIPS)* **2020**, *33*, 1877–1901.
35. Bengio, Y.; Courville, A.; Vincent, P. Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.* **2013**, *35*, 1798–1828. [CrossRef]
36. Lake, B.M.; Ullman, T.D.; Tenenbaum, J.B.; Gershman, S.J. Building machines that learn and think like people. *Behav. Brain Sci.* **2017**, *40*, e253. [CrossRef]
37. Weidinger, L.; Mellor, J.; Rauh, M.; Griffin, C.; Uesato, J.; Huang, P.S.; Cheng, M.; Glaese, M.; Balle, B.; Kasirzadeh, A.; et al. Ethical and social risks of harm from language models. *arXiv* **2021**, arXiv:2112.04359. [CrossRef]
38. Bubeck, S.; Chandrasekaran, V.; Eldan, R.; Gehrke, J.; Horvitz, E.; Kamar, E.; Lee, P.; Lee, Y.T.; Li, Y.; Lundberg, S.; et al. Sparks of artificial general intelligence: Early experiments with GPT-4. *arXiv* **2023**, arXiv:2303.12712. [CrossRef]
39. Zellers, R.; Holtzman, A.; Rashkin, H.; Bisk, Y.; Farhadi, A.; Roesner, F.; Choi, Y. Defending against neural fake news. In Proceedings of the Advances in Neural Information Processing Systems (NeurIPS), Vancouver, BC, Canada, 8–14 December 2019; Volume 32.
40. Post, M.; Vilar, D. Fast lexically constrained decoding with dynamic beam allocation for neural machine translation. *arXiv* **2018**, arXiv:1804.06609. [CrossRef]
41. Houlsby, N.; Giurgiu, A.; Jastrzebski, S.; Morrone, B.; De Laroussilhe, Q.; Gesmundo, A.; Attariyan, M.; Gelly, S. Parameter-efficient transfer learning for NLP. In *International Conference on Machine Learning*; PMLR: Baltimore, MD, USA, 2023; pp. 2790–2799.

42. Geirhos, R.; Jacobsen, J.H.; Michaelis, C.; Zemel, R.; Brendel, W.; Bethge, M.; Wichmann, F.A. Shortcut learning in deep neural networks. *Nat. Mach. Intell.* **2020**, *2*, 665–673. [CrossRef]
43. Vatsal, S.; Singh, A.; Tafreshi, S. Can GPT Improve the State of Prior Authorization Via Guideline Based Automated Question Answering? In *AI for Health Equity and Fairness*; Springer: Cham, Switzerland, 2024; pp. 147–158.
44. Hu, Y.; Chen, Q.; Du, J.; Peng, X.; Keloth, V.K.; Zuo, X.; Zhou, Y.; Li, Z.; Jiang, X.; Lu, Z.; et al. Improving large language models for clinical named entity recognition via prompt engineering. *J. Am. Med. Inform. Assoc.* **2024**, *31*, 1812–1820. [CrossRef]
45. Boyle, T. Medical Transcriptions. Available online: https://www.kaggle.com/datasets/tboyle10/medicaltranscriptions (accessed on 26 December 2024).
46. Centers for Disease Control and Prevention; U.S. Food and Drug Administration. Vaccine Adverse Event Reporting System (VAERS). Available online: https://vaers.hhs.gov/data/datasets.html (accessed on 26 December 2024).
47. Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Xia, F.; Chi, E.; Le Q.V.; Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. *Adv. Neural Inf. Process. Syst. (NeurIPS)* **2022**, *35*, 24824–24837.
48. Fu, Y.; Peng, H.; Sabharwal, A.; Clark, P.; Khot, T. Complexity-based prompting for multi-step reasoning. *arXiv* **2022**, arXiv:2210.00720.
49. Zhou, Y.; Geng, X.; Shen, T.; Tao, C.; Long, G.; Lou, J.G.; Shen, J. Thread of thought unraveling chaotic contexts. *arXiv* **2023**, arXiv:2311.08734. [CrossRef]
50. Li, X.; Zhao, R.; Chia, Y.K.; Ding, B.; Joty, S.; Poria, S.; Bing, L. Chain-of-knowledge: Grounding large language models via dynamic knowledge adapting over heterogeneous sources. *arXiv* **2023**, arXiv:2305.13269.
51. Li, C.; Liang, J.; Zeng, A.; Chen, X.; Hausman, K.; Sadigh, D.; Levine, S.; Fei-Fei, L.; Xia, F.; Ichter, B. Chain of code: Reasoning with a language model-augmented code emulator. *arXiv* **2023**, arXiv:2312.04474. [CrossRef]
52. Zhao, X.; Li, M.; Lu, W.; Weber, C.; Lee, J.H.; Chu, K.; Wermter, S. Enhancing zero-shot chain-of-thought reasoning in large language models through logic. *arXiv* **2023**, arXiv:2309.13339.
53. Bao, S.; Li, T.; Cao, B. Chain-of-event prompting for multi-document summarization by large language models. *Int. J. Web Inf. Syst.* **2024**, *20*, 229–247. [CrossRef]
54. Wang, Z.; Zhang, H.; Li, C.L.; Eisenschlos, J.M.; Perot, V.; Wang, Z.; Miculicich, L.; Fujii, Y.; Shang, J.; Lee, C.Y.; et al. Chain-of-table: Evolving tables in the reasoning chain for table understanding. *arXiv* **2024**, arXiv:2401.04398.
55. Wang, X.; Wei, J.; Schuurmans, D.; Le, Q.; Chi, E.; Narang, S.; Chowdhery, A.; Zhou, D. Self-consistency improves chain of thought reasoning in language models. *arXiv* **2022**, arXiv:2203.11171.
56. Chia, Y.K.; Chen, G.; Tuan, L.A.; Poria, S.; Bing, L. Contrastive chain-of-thought prompting. *arXiv* **2023**, arXiv:2311.09277.
57. Liu, X.; Pang, T.; Fan, C. Federated prompting and chain-of-thought reasoning for improving LLMs answering. In *International Conference on Knowledge Science, Engineering and Management*; Springer Nature: Cham, Switzerland, 2023; pp. 3–11.
58. Yao, S.; Yu, D.; Zhao, J.; Shafran, I.; Griffiths, T.; Cao, Y.; Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models. *Adv. Neural Inf. Process. Syst. (NeurIPS)* **2024**, *36*, 11809–11822.
59. Jung, J.; Qin, L.; Welleck, S.; Brahman, F.; Bhagavatula, C.; Le Bras, R.; Choi, Y. Maieutic prompting: Logically consistent reasoning with recursive explanations. *arXiv* **2022**, arXiv:2205.11822. [CrossRef]
60. Wang, L.; Xu, W.; Lan, Y.; Hu, Z.; Lan, Y.; Lee, R.K.W.; Lim, E.P. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv* **2023**, arXiv:2305.04091.
61. Chen, W.; Ma, X.; Wang, X.; Cohen, W.W. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv* **2022**, arXiv:2211.12588.
62. Hu, H.; Lu, H.; Zhang, H.; Song, Y.Z.; Lam, W.; Zhang, Y. Chain-of-symbol prompting elicits planning in large language models. *arXiv* **2023**, arXiv:2305.10276.
63. Li, J.; Li, G.; Li, Y.; Jin, Z. Structured chain-of-thought prompting for code generation. *ACM Trans. Softw. Eng. Methodol.* **2023**, *34*, 1–23. [CrossRef]
64. Fei, H.; Li, B.; Liu, Q.; Bing, L.; Li, F.; Chua, T.S. Reasoning implicit sentiment with chain-of-thought prompting. *arXiv* **2023**, arXiv:2305.11255.
65. Singhal, K.; Tu, T.; Gottweis, J.; Sayres, R.; Wulczyn, E.; Hou, L.; Clark, K.; Pfohl, S.; Cole-Lewis, H.; Neal, D.; et al. Towards expert-level medical question answering with large language models. *arXiv* **2023**, arXiv:2305.09617. [CrossRef]
66. Zhang, Z.; Zhang, A.; Li, M.; Smola, A. Automatic chain of thought prompting in large language models. *arXiv* **2022**, arXiv:2210.03493. [CrossRef]
67. Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; Cao, Y. ReAct: Synergizing reasoning and acting in language models. *arXiv* **2022**, arXiv:2210.03629.
68. Diao, S.; Wang, P.; Lin, Y.; Pan, R.; Liu, X.; Zhang, T. Active prompting with chain-of-thought for large language models. *arXiv* **2023**, arXiv:2302.12246.
69. Imani, S.; Du, L.; Shrivastava, H. MathPrompter: Mathematical reasoning using large language models. *arXiv* **2023**, arXiv:2303.05398. [CrossRef]

70. Yasunaga, M.; Chen, X.; Li, Y.; Pasupat, P.; Leskovec, J.; Liang, P.; Chi, E.H.; Zhou, D. Large language models as analogical reasoners. *arXiv* **2023**, arXiv:2310.01714.
71. Shao, Z.; Gong, Y.; Shen, Y.; Huang, M.; Duan, N.; Chen, W. Synthetic Prompting: Generating Chain-of-Thought Demonstrations for Large Language Models. *arXiv* **2023**, arXiv:2302.00618.
72. Weston, J.; Sukhbaatar, S. System 2 Attention (is something you might need too). *arXiv* **2023**, arXiv:2311.11829. [CrossRef]
73. Wang, Y.; Zhao, Y. Metacognitive prompting improves understanding in large language models. *arXiv* **2023**, arXiv:2308.05342.
74. Zhou, D.; Schärli, N.; Hou, L.; Wei, J.; Scales, N.; Wang, X.; Schuurmans, D.; Cui, C.; Bousquet, O.; Le, Q.; et al. Least-to-most prompting enables complex reasoning in large language models. *arXiv* **2022**, arXiv:2205.10625.
75. Khot, T.; Trivedi, H.; Finlayson, M.; Fu, Y.; Richardson, K.; Clark, P.; Sabharwal, A. Decomposed prompting: A modular approach for solving complex tasks. *arXiv* **2022**, arXiv:2210.02406.
76. Gao, L.; Madaan, A.; Zhou, S.; Alon, U.; Liu, P.; Yang, Y.; Callan, J.; Neubig, G. PAL: Program-aided Language Models. *arXiv* **2022**, arXiv:2211.10435.
77. Cheng, Z.; Xie, T.; Shi, P.; Li, C.; Nadkarni, R.; Hu, Y.; Xiong, C.; Radev, D.; Ostendorf, M.; Zettlemoyer, L.; et al. Binding language models in symbolic languages. *arXiv* **2022**, arXiv:2210.02875.
78. Ye, Y.; Hui, B.; Yang, M.; Li, B.; Huang, F.; Li, Y. Large language models are versatile decomposers: Decompose evidence and questions for table-based reasoning. *arXiv* **2023**, arXiv:2301.13808. [CrossRef]
79. Lewis, P.; Perez, E.; Piktus, A.; Petroni, F.; Karpukhin, V.; Goyal, N.; Küttler, H.; Lewis, M.; Yih, W.-t.; Rocktäschel, T.; et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. *Adv. Neural Inf. Process. Syst. (NeurIPS)* **2020**, *33*, 9459–9474.
80. Béchard, P.; Ayala, O.M. Reducing hallucination in structured outputs via Retrieval-Augmented Generation. *arXiv* **2024**, arXiv:2404.08189. [CrossRef]
81. Dixit, P.; Oates, T. SBI-RAG: Enhancing math word problem solving for students through schema-based instruction and retrieval-augmented generation. *arXiv* **2024**, arXiv:2410.13293.
82. Matsumoto, N.; Moran, J.; Choi, H.; Hernandez, M.E.; Venkatesan, M.; Wang, P.; Moore, J.H. KRAGEN: A knowledge graph-enhanced RAG framework for biomedical problem solving using large language models. *Bioinformatics* **2024**, *40*, btae353. [CrossRef]
83. Liu, X.; Wang, R.; Song, Y.; Kong, L. GRAM: Generative Retrieval Augmented Matching of Data Schemas in the Context of Data Security. In Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining KDD '24, Barcelona, Spain, 25–29 August 2024; pp. 5476–5486.
84. Chen, S.-A.; Miculicich, L.; Eisenschlos, J.M.; Wang, Z.; Wang, Z.; Chen, Y.; Fujii, Y.; Lin, H.T.; Lee, C.Y.; Pfister, T. TableRAG: Million-token table understanding with language models. *arXiv* **2024**, arXiv:2410.04739.
85. Yao, Z.; Qi, W.; Pan, L.; Cao, S.; Hu, L.; Liu, W.; Hou, L.; Li, J. Seakr: Self-aware knowledge retrieval for adaptive retrieval augmented generation. *arXiv* **2024**, arXiv:2406.19215.
86. Asai, A.; Wu, Z.; Wang, Y.; Sil, A.; Hajishirzi, H. Self-RAG: Self-reflective retrieval augmented generation. In Proceedings of the NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following, New Orleans, LA, USA, 15 December 2023.
87. Wang, Z.; Wang, Z.; Le, L.; Zheng, H.S.; Mishra, S.; Perot, V.; Zhang, Y.; Mattapalli, A.; Taly, A.; Shang, J.; et al. Speculative RAG: Enhancing retrieval augmented generation through drafting. *arXiv* **2024**, arXiv:2407.08223. [CrossRef]
88. Xu, R.; Liu, H.; Nag, S.; Dai, Z.; Xie, Y.; Tang, X.; Luo, C.; Li, Y.; Ho, J.C.; Yang, C.; et al. SimRAG: Self-Improving Retrieval-Augmented Generation for Adapting Large Language Models to Specialized Domains. *arXiv* **2024**, arXiv:2410.17952.
89. Li, X.; Xu, W.; Zhao, R.; Jiao, F.; Joty, S.; Bing, L. Can We Further Elicit Reasoning in LLMs? Critic-Guided Planning with Retrieval-Augmentation for Solving Challenging Tasks. *arXiv* **2024**, arXiv:2410.01428.
90. Hu, M.; Zong, L.; Wang, H.; Zhou, J.; Li, J.; Gao, Y.; Wong, K.F.; Li, Y.; King, I. SeRTS: Self-Rewarding Tree Search for Biomedical Retrieval-Augmented Generation. In *Findings of ACL: EMNLP 2024*; Association for Computational Linguistics: Miami, FL, USA, 2024; pp. 1321–1335.
91. Qu, C.; Yang, L.; Qiu, M.; Croft, W.B.; Zhang, Y.; Iyyer, M. BERT with history answer embedding for conversational question answering. In Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval SIGIR '19, Paris, France, 21–25 July 2019; pp. 1133–1136.
92. Gutiérrez, B.J.; Shu, Y.; Gu, Y.; Yasunaga, M.; Su, Y. HippoRAG: Neurobiologically Inspired Long-Term Memory for Large Language Models. *arXiv* **2024**, arXiv:2405.14831.
93. Ho, X.; Nguyen, A.K.D.; Sugawara, S.; Aizawa, A. Constructing a multi-hop QA dataset for comprehensive evaluation of reasoning steps. In Proceedings of the COLING 2020, Online, 8–13 December 2020; pp. 6609–6625.
94. Yang, Z.; Qi, P.; Zhang, S.; Bengio, Y.; Cohen, W.W.; Salakhutdinov, R.; Manning, C.D. HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering. In Proceedings of the EMNLP 2018, Brussels, Belgium, 31 October–4 November 2018; pp. 2369–2380.

95. Sarmah, B.; Mehta, D.; Hall, B.; Rao, R.; Patel, S.; Pasquali, S. HybridRAG: Integrating Knowledge Graphs and Vector Retrieval Augmented Generation for Efficient Information Extraction. In *ACM International Conference on AI in Finance (ICAIF '24)*; ACM: New York, NY, USA, 2024; pp. 608–616.
96. Hu, Y.; Lei, Z.; Zhang, Z.; Pan, B.; Ling, C.; Zhao, L. GRAG: Graph Retrieval-Augmented Generation. *arXiv* **2024**, arXiv:2405.16506. [CrossRef]
97. Cai, Y.; Guo, Z.; Pei, Y.; Bian, W.; Zheng, W. SimGRAG: Leveraging Similar Subgraphs for Knowledge Graphs Driven Retrieval-Augmented Generation. *arXiv* **2024**, arXiv:2412.15272.
98. Wang, W.; Dong, L.; Cheng, H.; Liu, X.; Yan, X.; Gao, J.; Wei, F. Augmenting language models with long-term memory. *Adv. Neural Inf. Process. Syst. (NeurIPS)* **2024**, *36*, 74530–74543.
99. Aadhithya, A.; Kumar, S.; Soman, K.P. Enhancing Long-Term Memory using Hierarchical Aggregate Tree for Retrieval Augmented Generation. *arXiv* **2024**, arXiv:2406.06124. [CrossRef]
100. Qian, H.; Zhang, P.; Liu, Z.; Mao, K.; Dou, Z. MemoRAG: Moving towards next-gen RAG via memory-inspired knowledge discovery. *arXiv* **2024**, arXiv:2409.05591.
101. Bai, Y.; Miao, Y.; Chen, L.; Wang, D.; Li, D.; Ren, Y.; Xie, H.; Yang, C.; Cai, X. Pistis-RAG: Enhancing Retrieval-Augmented Generation with Human Feedback. *arXiv* **2024**, arXiv:2407.00072.
102. Gan, C.; Yang, D.; Hu, B.; Zhang, H.; Li, S.; Liu, Z.; Shen, Y.; Ju, L.; Zhang, Z.; Gu, J.; et al. Similarity is Not All You Need: Endowing Retrieval Augmented Generation with Multi Layered Thoughts. *arXiv* **2024**, arXiv:2405.19893. [CrossRef]
103. Jiang, J.; Chen, J.; Li, J.; Ren, R.; Wang, S.; Zhao, W.X.; Song, Y.; Zhang, T. RAG-Star: Enhancing Deliberative Reasoning with Retrieval Augmented Verification and Refinement. *arXiv* **2024**, arXiv:2412.12881. [CrossRef]
104. Cho, J.; Mahata, D.; Irsoy, O.; He, Y.; Bansal, M. M3DoCRAG: Multi-modal retrieval is what you need for multi-page multi-document understanding. *arXiv* **2024**, arXiv:2411.04952.
105. Yu, S.; Tang, C.; Xu, B.; Cui, J.; Ran, J.; Yan, Y.; Liu, Z.; Wang, S.; Han, X.; Liu, Z.; et al. VisRAG: Vision-based retrieval-augmented generation on multi-modality documents. *arXiv* **2024**, arXiv:2410.10594.
106. Li, Y.; Li, Y.; Wang, X.; Jiang, Y.; Zhang, Z.; Zheng, X.; Wang, H.; Yu, P.S.; Huang, F.; Zhou, J. Benchmarking Multimodal Retrieval Augmented Generation with Dynamic VQA Dataset and Self-adaptive Planning Agent. *arXiv* **2024**, arXiv:2411.02937. [CrossRef]
107. Dhuliawala, S.; Komeili, M.; Xu, J.; Raileanu, R.; Li, X.; Celikyilmaz, A.; Weston, J. Chain-of-verification reduces hallucination in large language models. *arXiv* **2023**, arXiv:2309.11495.
108. Zhao, R.; Li, X.; Joty, S.; Qin, C.; Bing, L. Verify-and-edit: A knowledge-enhanced chain-of-thought framework. *arXiv* **2023**, arXiv:2305.03268.
109. Zhai, Y.; Bai, H.; Lin, Z.; Pan, J.; Tong, S.; Zhou, Y.; Suhr, A.; Xie, S.; LeCun, Y.; Ma, Y.; et al. Fine-Tuning Large Vision-Language Models as Decision-Making Agents via reinforcement learning. *arXiv* **2024**, arXiv:2405.10292.
110. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*, 2nd ed.; MIT Press: Cambridge, MA, USA, 2018.
111. Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; Klimov, O. Proximal policy optimization algorithms. *arXiv* **2017**, arXiv:1707.06347. [CrossRef]
112. Ouyang, L.; Wu, J.; Jiang, X.; Almeida, D.; Wainwright, C.; Mishkin, P.; Zhang, C.; Agarwal, S.; Slama, K.; Ray, A.; et al. Training language models to follow instructions with human feedback. *Adv. Neural Inf. Process. Syst. (NeurIPS)* **2022**, *35*, 27730–27744.
113. Shen, W.; Zheng, R.; Zhan, W.; Zhao, J.; Dou, S.; Gui, T.; Zhang, Q.; Huang, X. Loose lips sink ships: Mitigating length bias in reinforcement learning from human feedback. *arXiv* **2023**, arXiv:2310.05199. [CrossRef]
114. Wang, B.; Zheng, R.; Chen, L.; Liu, Y.; Dou, S.; Huang, C.; Shen, W.; Jin, S.; Zhou, E.; Shi, C.; et al. Secrets of RLHF in large language models Part II: Reward modeling. *arXiv* **2024**, arXiv:2401.06080. [CrossRef]
115. Havrilla, A.; Zhuravinskyi, M.; Phung, D.; Tiwari, A.; Tow, J.; Biderman, S.; Anthony, Q.; Castricato, L. trlX: A framework for large scale reinforcement learning from human feedback. In Proceedings of the EMNLP 2023, Singapore, 6–10 December 2023; pp. 8578–8595.
116. Cui, G.; Yuan, L.; Ding, N.; Yao, G.; Zhu, W.; Ni, Y.; Xie, G.; Liu, Z.; Sun, M. Ultrafeedback: Boosting language models with high-quality feedback. *arXiv* **2023**, arXiv:2310.01377.
117. Liu, C.Y.; Zeng, L.; Liu, J.; Yan, R.; He, J.; Wang, C.; Yan, S.; Liu, Y.; Zhou, Y. Skywork-reward: Bag of tricks for reward modeling in LLMs. *arXiv* **2024**, arXiv:2410.18451. [CrossRef]
118. Ivison, H.; Wang, Y.; Pyatkin, V.; Lambert, N.; Peters, M.; Dasigi, P.; Jang, J.; Wadden, D.; Smith, N.A.; Beltagy, I.; et al. Camels in a changing climate: Enhancing LM adaptation with Tulu 2. *arXiv* **2023**, arXiv:2311.10702. [CrossRef]
119. Li, L.; Chai, Y.; Wang, S.; Sun, Y.; Tian, H.; Zhang, N.; Wu, H. Tool-augmented reward modeling. *arXiv* **2023**, arXiv:2310.01045.
120. Munos, R.; Valko, M.; Calandriello, D.; Azar, M.G.; Rowland, M.; Guo, Z.D.; Tang, Y.; Geist, M.; Mesnard, T.; Michi, A.; et al. Nash learning from human feedback. *arXiv* **2023**, arXiv:2312.00886. [CrossRef]
121. Gao, L.; Schulman, J.; Hilton, J. Scaling laws for reward model overoptimization. In Proceedings of the International Conference on Machine Learning (ICML), Honolulu, HI, USA, 23–19 July 2023; pp. 10835–10866.

122. Kaufmann, T.; Weng, P.; Bengs, V.; Hüllermeier, E. A survey of reinforcement learning from human feedback. *arXiv* **2023**, arXiv:2312.14925.
123. Lee, H.; Phatale, S.; Mansoor, H.; Lu, K.R.; Mesnard, T.; Ferret, J.; Bishop, C.; Hall, E.; Carbune, V.; Rastogi, A. RLAIF vs. RLHF: Scaling reinforcement learning from human feedback with AI feedback. *arXiv* **2023**, arXiv:2309.00267.
124. Li, A.; Xiao, Q.; Cao, P.; Tang, J.; Yuan, Y.; Zhao, Z.; Chen, X.; Zhang, L.; Li, X.; Yang, K.; et al. HRLAIF: Improvements in helpfulness and harmlessness in open-domain reinforcement learning from AI feedback. *arXiv* **2024**, arXiv:2403.08309. [CrossRef]
125. Xu, Z.; Jiang, F.; Niu, L.; Deng, Y.; Poovendran, R.; Choi, Y.; Lin, B.Y. Magpie: Alignment Data Synthesis from Scratch by Prompting Aligned LLMs with Nothing. *arXiv* **2024**, arXiv:2406.08464. [CrossRef]
126. Wang, Z.; Bukharin, A.; Delalleau, O.; Egert, D.; Shen, G.; Zeng, J.; Kuchaiev, O.; Dong, Y. Helpsteer2-preference: Complementing ratings with preferences. *arXiv* **2024**, arXiv:2410.01257.
127. Du, Y.; Watkins, O.; Wang, Z.; Colas, C.; Darrell, T.; Abbeel, P.; Gupta, A.; Andreas, J. Guiding pretraining in reinforcement learning with large language models. In Proceedings of the International Conference on Machine Learning (ICML), Honolulu, HI, USA, 23–29 July 2023; pp. 8657–8677.
128. Kwon, M.; Xie, S.M.; Bullard, K.; Sadigh, D. Reward design with language models. *arXiv* **2023**, arXiv:2303.00001. [CrossRef]
129. Ma, Y.J.; Liang, W.; Wang, G.; Huang, D.A.; Bastani, O.; Jayaraman, D.; Zhu, Y.; Fan, L.; Anandkumar, A. Eureka: Human-level reward design via coding large language models. *arXiv* **2023**, arXiv:2310.12931.
130. Song, J.; Zhou, Z.; Liu, J.; Fang, C.; Shu, Z.; Ma, L. Self-refined large language model as automated reward function designer for deep reinforcement learning in robotics. *arXiv* **2023**, arXiv:2309.06687. [CrossRef]
131. Yuan, W.; Pang, R.Y.; Cho, K.; Sukhbaatar, S.; Xu, J.; Weston, J. Self-rewarding language models. *arXiv* **2024**, arXiv:2401.10020.
132. Wang, H.; Zariphopoulou, T.; Zhou, X. Exploration versus exploitation in reinforcement learning: A stochastic control approach. *arXiv* **2018**, arXiv:1812.01552. [CrossRef]
133. Dann, C.; Mansour, Y.; Mohri, M.; Sekhari, A.; Sridharan, K. Guarantees for epsilon-greedy reinforcement learning with function approximation. In Proceedings of the ICML, Baltimore, MD, USA, 17–23 July 2022; pp. 4666–4689.
134. Tokic, M. Adaptive ε-greedy exploration in reinforcement learning based on value differences. In *Annual Conference on Artificial Intelligence*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 203–210.
135. Cesa-Bianchi, N.; Gentile, C.; Lugosi, G.; Neu, G. Boltzmann exploration done right. In Proceedings of the Advances in Neural Information Processing Systems (NeurIPS), Long Beach, CA, USA, 4–9 December 2017; Volume 30.
136. Ma, R.; Luijkx, J.; Ajanovic, Z.; Kober, J. ExploRLLM: Guiding Exploration in Reinforcement Learning with Large Language Models. *arXiv* **2024**, arXiv:2403.09583. [CrossRef]
137. Zhao, Q.; Fu, H.; Sun, C.; Konidaris, G. EPO: Hierarchical LLM agents with environment preference optimization. *arXiv* **2024**, arXiv:2408.16090. [CrossRef]
138. Nguyen, H.-T.; Satoh, K. Balancing Exploration and Exploitation in LLM using Soft RLLF for Enhanced Negation Understanding. *arXiv* **2024**, arXiv:2403.01185. [CrossRef]
139. Yang, F.; Zhao, P.; Wang, Z.; Wang, L.; Zhang, J.; Garg, M.; Lin, Q.; Rajmohan, S.; Zhang, D. Empower large language model to perform better on industrial domain-specific question answering. *arXiv* **2023**, arXiv:2305.11541. [CrossRef]
140. Rafailov, R.; Sharma, A.; Mitchell, E.; Manning, C.D.; Ermon, S.; Finn, C. Direct preference optimization: Your language model is secretly a reward model. *Adv. Neural Inf. Process. Syst. (NeurIPS)* **2024**, *36*, 53728–53741.
141. Pal, A.; Karkhanis, D.; Dooley, S.; Roberts, M.; Naidu, S.; White, C. Smaug: Fixing failure modes of preference optimisation with DPO-positive. *arXiv* **2024**, arXiv:2402.13228. [CrossRef]
142. Wu, J.; Xie, Y.; Yang, Z.; Wu, J.; Gao, J.; Ding, B.; Wang, X.; He, X. β-DPO: Direct Preference Optimization with Dynamic β. *arXiv* **2024**, arXiv:2407.08639.
143. Kim, D.; Kim, Y.; Song, W.; Kim, H.; Kim, Y.; Kim, S.; Park, C. sDPO: Don't Use Your Data All at Once. *arXiv* **2024**, arXiv:2403.19270.
144. Yang, Z.; Wan, F.; Zhong, L.; Shi, T.; Quan, X. Weighted-Reward Preference Optimization for Implicit Model Fusion. *arXiv* **2024**, arXiv:2412.03187. [CrossRef]
145. Chowdhury, S.R.; Kini, A.; Natarajan, N. Provably robust DPO: Aligning language models with noisy feedback. *arXiv* **2024**, arXiv:2403.00409. [CrossRef]
146. Azar, M.G.; Guo, Z.D.; Piot, B.; Munos, R.; Rowland, M.; Valko, M.; Calandriello, D. A general theoretical paradigm to understand learning from human preferences. In Proceedings of the AISTATS, Valencia, Spain, 2–4 May 2024; pp. 4447–4455.
147. Sun, H.; Shen, Y.; Ton, J.F. Rethinking Bradley-Terry Models in Preference-Based Reward Modeling: Foundations, Theory, and Alternatives. *arXiv* **2024**, arXiv:2411.04991.
148. Wu, Y.; Sun, Z.; Yuan, H.; Ji, K.; Yang, Y.; Gu, Q. Self-play preference optimization for language model alignment. *arXiv* **2024**, arXiv:2405.00675. [CrossRef]
149. Wang, C.; Zhao, Z.; Zhu, C.; Sankararaman, K.A.; Valko, M.; Cao, X.; Chen, Z.; Khabsa, M.; Chen, Y.; Ma, H.; et al. Preference optimization with multi-sample comparisons. *arXiv* **2024**, arXiv:2410.12138. [CrossRef]

150. Fisch, A.; Eisenstein, J.; Zayats, V.; Agarwal, A.; Beirami, A.; Nagpal, C.; Shaw, P.; Berant, J. Robust preference optimization through reward model distillation. *arXiv* **2024**, arXiv:2405.19316. [CrossRef]
151. Dong, Y.; Luo, K.; Jiang, X.; Jin, Z.; Li, G. PACE: Improving Prompt with Actor-Critic Editing for Large Language Model. *arXiv* **2023**, arXiv:2308.10088. [CrossRef]
152. Ziegler, D.M.; Stiennon, N.; Wu, J.; Brown, T.B.; Radford, A.; Amodei, D.; Christiano, P.; Irving, G. Fine-tuning language models from human preferences. *arXiv* **2019**, arXiv:1909.08593.
153. Yao, W.; Mi, H.; Yu, D. HDFlow: Enhancing LLM Complex Problem-Solving with Hybrid Thinking and Dynamic Workflows. *arXiv* **2024**, arXiv:2409.17433. [CrossRef]
154. Liu, G.; Ji, K.; Zheng, R.; Wu, Z.; Dun, C.; Gu, Q.; Yan, L. Enhancing Multi-Step Reasoning Abilities of Language Models through Direct Q-Function Optimization. *arXiv* **2024**, arXiv:2410.09302.
155. Rumelhart, D.E. Schemata: The building blocks of cognition. In *Theoretical Issues in Reading Comprehension*; Spiro, R.J., Bruce, B.C., Brewer, W.F., Eds.; Lawrence Erlbaum: Hillsdale, NJ, USA, 1980.
156. Liu, Z.; Kitouni, O.; Nolte, N.S.; Michaud, E.; Tegmark, M.; Williams, M. Towards understanding grokking: An effective theory of representation learning. *Adv. Neural Inf. Process. Syst. (NeurIPS)* **2022**, *35*, 34651–34663.
157. d'Avila Garcez, A.; Lamb, L.C. Neurosymbolic AI: The 3rd Wave. *arXiv* **2020**, arXiv:2012.05876. Available online: https://arxiv.org/abs/2012.05876 (accessed on 24 October 2025). [CrossRef]
158. Mao, J.; Gan, C.; Kohli, P.; Tenenbaum, J.B.; Wu, J. The Neuro-Symbolic Concept Learner: Interpreting Scenes, Words, and Sentences From Natural Supervision. *arXiv* **2019**, arXiv:1904.12584. Available online: https://arxiv.org/abs/1904.12584 (accessed on 24 October 2025). [CrossRef]
159. Xu, H.; Hu, J.; Zhang, K.; Yu, L.; Tang, Y.; Song, X.; Duan, Y.; Ai, L.; Shi, B. SEDM: Scalable Self-Evolving Distributed Memory for Agents. *arXiv* **2025**, arXiv:2509.09498v3. Available online: https://arxiv.org/html/2509.09498v3 (accessed on 24 October 2025). [CrossRef]
160. Zhu, Z.; Liao, Y.; Xu, C.; Guan, Y.; Wang, Y.; Wang, Y. RA^{2}FD: Distilling Faithfulness into Efficient Dialogue Systems. In Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP), Miami, FL, USA, 12–16 November 2024; Association for Computational Linguistics: Miami, FL, USA, 2024; pp. 12304–12317. [CrossRef]
161. Lu, S.; Wang, L.; Wen, S.; Wang, Z.; Zhang, H. FedDTRE: Federated Dialogue Generation Models Powered by Trustworthiness Evaluation. *arXiv* **2025**, arXiv:2510.08058. [CrossRef]
162. Sun, W.; Cai, H.; Chen, H.; Ren, P.; Chen, Z.; de Rijke, M.; Ren, Z. Answering ambiguous questions via iterative prompting. *arXiv* **2023**, arXiv:2307.03897. [CrossRef]
163. Ginsburg, L.R.; Hoben, M.; Easterbrook, A.; Anderson, R.A.; Estabrooks, C.A.; Norton, P.G. Fidelity is not easy! Challenges and guidelines for assessing fidelity in complex interventions. *Trials* **2021**, *22*, 372. [CrossRef]
164. van Stein, N.; Bäck, T. Llamea: A Large Language Model Evolutionary Algorithm for Automatically Generating Metaheuristics. *IEEE Trans. Evol. Comput.* **2024**, *29*, 331–345. [CrossRef]
165. Malone, T.; Bernstein, M. *Handbook of Collective Intelligence*; The MIT Press: Cambridge, MA, USA, 2015.
166. Woolley, A.W.; Aggarwal, I.; Malone, T.W. Collective Intelligence in Teams and Organizations. In *Handbook of Collective Intelligence*; Malone, T., Bernstein, M., Eds.; The MIT Press: Cambridge, MA, USA, 2015.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.