Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Adaptive Universal Prompt Optimizer: A Multi-Stage Prompt Engineering Framework Using the Model Context Protocol

Arpit Kale

arpitkale2005@gmail.com

G H Raisoni College of Engineering

Arya Tiwari

G H Raisoni College of Engineering

Dhanistha Wadettiwar

G H Raisoni College of Engineering

Dhanshri Kubade

G H Raisoni College of Engineering

Hansaraj Wankhede

G H Raisoni College of Engineering

Research Article

Keywords: prompt engineering, large language models, prompt optimization, Model Context Protocol, rule-based evaluation, text classification

Posted Date: November 17th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11736368/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Adaptive Universal Prompt Optimizer: A Multi-Stage Prompt Engineering Framework Using the Model Context Protocol

Arpit Kale
G H Raisoni College of Engineering
Nagpur, India

Arya Tiwari
G H Raisoni College of Engineering
Nagpur, India

Dhanistha Wadettiwar
G H Raisoni College of Engineering
Nagpur, India

Dhanshri Kubade
G H Raisoni College of Engineering
Nagpur, India

Dr. Hansaraj Wankhede
G H Raisoni College of Engineering
Nagpur, India

***Abstract*— Large language models provide more accurate responses when given well-structured and clear prompts. However, most users' prompts are brief and ambiguous. Writing effective prompts can be time-consuming and requires skill. This paper describes the Adaptive Universal Prompt Optimizer, a lightweight python program that takes a user input prompt and produces an optimized system prompt and well-structured user prompt. It identifies the task type using an ordered keyword classifier before assigning the model an appropriate expert role. The prompt is then created using techniques from the literature. This includes APET for strategic improvements, PICCO for structure, MetaSPO for the system prompt and OPRO for iterative improvements. A rule-based scoring system (APEX) tests each prompt on four aspects: Assessment, Performance, Execution and Excellence. The system asks up to three brief questions to determine the desired level of detail, content and audience. The system operates either as a command line application or as a Model Context Protocol server. The system was tested with 60 prompts across 6 different task types. The ordered keyword classifier correctly identified the task type for 93.3% of prompts. The average APEX score improved from 36.4 to 70.9. When the user added details regarding the desired level of detail and audience, the score improved further to 74.5. Optimization occurs in under 0.1ms. No prompts scored above the pass mark (85) and after one iteration of improvements the improvements were no longer beneficial. Overall, this shows the system is quick and has potential but further tuning is required to improve the scoring system and the rules applied to improve.**

Keywords—prompt engineering, large language models, prompt optimization, Model Context Protocol, rule-based evaluation, text classification

I. INTRODUCTION

A. Background

Large language models (LLMs) are neural networks trained on vast amounts of text [1]. They can be trained further to understand instructions [3]. With these additional capabilities, they can now be used to answer questions, write code, summarise papers and explain complex topics. They are commonly used by students, teachers, engineers and business people. The majority of people have only one method of controlling the LLM, this is by using the prompt. The prompt is the text that is typed in, so the better the prompt usually the better the answer given.

Writing a good prompt is hard. Zamfirescu-Pereira et al. [4] studied users and found that people without a technical background often write prompts by trial and error, and they give up when the first answer is bad. Other studies show that even a small change in words or format can change the model's output a lot [5], [6]. For this reason, prompt engineering has become a fast-growing field [7], [8], [9].

B. Problem Statement

When given a short prompt such as "tell me about DSA", the model lacks information about its audience or the depth of detail required in the response. This typically leads to generic, verbose responses which do not provide enough useful information. While there are many automatic prompt improvement techniques [10] [11] [12] [13] [14] [15], they tend to require large amounts of labelled data, multiple calls to the model or training. These are focussed on helping researchers optimise a single prompt for their specific task. However, we want to help students achieve a better response to a new question.

The Model Context Protocol (MCP) [16] provides a standardized method for connecting external tools to AI assistants. This means that prompt improvement can be provided as an internal tool to the AI assistant. Our aim is to develop a simple, rule-based tool that can quickly improve any raw prompt and provide reasoning for its scoring whilst being compatible with common assistants using the MCP.

Objectives

The objectives of this work are as follows:

- To build a pipeline that takes any raw request and gives back an improved system prompt and user prompt.
- To find the type of task on its own, and to pick a matching expert role and prompt structure.
- To learn what the user wants about depth, content, and audience by asking only a few simple questions.
- To score each prompt with a clear set of rules, and to improve the prompt when the score is low.
- To make the system work both as a command-line program and as an MCP server.
- To measure the quality, speed, and limits of the system on a set of test requests.

C. Contributions

This paper makes the following contributions:

- A general, rule-based optimizer that puts a request into one of six task types and builds a full prompt without any extra model call.

- An 11-stage workflow that brings five published ideas (APET, PICCO, MetaSPO, APEX, and OPRO) together in one working tool.
- A clear scorer with four parts, a pass mark, and an improvement loop that uses the scorer's list of problems.
- An MCP server with 17 tools, plus command-line and interactive modes.
- A test on 60 requests that looks at classification accuracy, score changes, the effect of user choices, the improvement loop, the pass mark, prompt length, and running time, along with a case study of eight real answers.

D. Organization of the Paper

The theory of the system is described in section II. Then related work is covered in section III showing the research gap. The description of the proposed system is given in section IV. Implementation details and cost analysis are given in section V. Section VI describes the setup of the testing. Section VII presents and discusses the results. The application of the system is discussed in section VIII. Ethical issues are discussed in section IX. The limitations of the system are listed in section X. Future work is presented in section XI. Finally section XII concludes the paper.

II. Background and Theory

A. Language Models and Prompts

Most modern LLMs use the transformer design [1]. The model reads a list of tokens and guesses the next token, one step at a time. A prompt is just the start of this list, and everything the model writes depends on it. Brown et al. [2] showed that large models can learn a new task from the prompt alone, without changing their weights. This is called in-context learning. It is the main reason why the words in a prompt matter so much: the prompt is the only place where the task is described.

B. System Prompt and User Prompt

Chat models usually take two kinds of input. The system prompt sets the general role, rules, and style of the assistant, and it stays the same for the whole conversation. The user prompt holds the actual request. Because the system prompt affects many answers, a good system prompt can help with many tasks [14]. Our system writes both parts. The system prompt holds the expert role and the quality rules. The user prompt holds the task, the structure, and the user's choices.

C. Common Prompting Techniques

Some simple methods are known to be effective. Zero shot prompting provides the model with only an instruction. Few-shot prompting works by providing the model with a small number of example solutions [2]. Chain of thought prompting causes the model to give reasoning steps and therefore improves their reasoning [17]. Even including a short phrase such as 'let's think step by step' was shown to help [18]. Self-consistency asks the model to give multiple answers and keeps track of the most common answer [19]. Tree of thoughts provides multiple reasoning chains and then chooses one [20]. Role prompting provides the model with a role to take on such as 'you are a senior teacher'. It has been shown that role playing can improve zero-shot reasoning [21]. However it has been shown that the benefit depends on the role taken on and the task [22]. Our system uses role prompting with step-by-step, but doesn't include any example solutions.

Other research has provided useful guidelines to create prompts. Prompts can be seen as a type of program [23]. Break complex instructions into smaller simpler ones [24]. White et al. have created a collection of prompts which can be reused [25]. Many rules have been identified such as specifying who you are addressing and having clear separators [26]. Many of these were applied directly to our templates.

D. Prompt Sensitivity

LLMs react to small details. Sclar et al. [5] showed that changes in spaces, separators, or capital letters can move a model's accuracy by many points. Mizrahi et al. [6] showed that one prompt is not enough to judge a model fairly. This is both a problem and a chance. It is a problem because users cannot guess how their words will change the result. It is a chance because careful wording can bring large gains for free.

E. Prompt Optimization as a Search Problem

Prompt optimization can be written as a search problem. Let M be a model, let x be an input taken from a task distribution D, and let f be a quality measure for the output. The best prompt is:

$$p^* = \arg\max_{p \in P} E_{x \sim D} [\, f(M(p, x)) \,] \quad (1)$$

where P is the set of all possible prompts. Methods differ in how they search this huge space. Some use gradients on continuous vectors [27], [28]. Some edit words [29] or use reinforcement learning [30]. Many recent methods ask an LLM to write new candidates [11], [10], [12]. All of them need a way to compute f, and this usually means a labeled dataset and many model calls. Our system replaces f with a fixed rubric that looks only at the prompt. This makes it very cheap, but it also means that the score measures the form of the prompt and not the true quality of the answer. We return to this point in Section X.

F. Rule-Based Text Classification

Text classification assigns labels to texts. There are two main types of text classifiers, rule based systems and those that use learning [31]. The rule based classifiers are quick to use as they require no training data and are easy to understand. However, they might fail to account for novel phrases. The learning based approaches such as BERT [32] and sentence embeddings [33] are better at understanding novel phrases but do require data and have a higher computational demand. In this work we chose to use a rule based classifier. This checks through ordered lists of keywords and assigns the label based on the first matching list. The accuracy of the classifier is shown in Section VII.

G. Learning Theory Behind Structured Answers

The use of a structured prompting system can help a human reader due to two theories in education. The updated Bloom taxonomy [34] provides a hierarchy of learning objectives. This is reflected by the 3 depth-levels in our question step, where the surface-level provides a quick summary (to understand) and the detailed-level seeks to analyse and judge. Cognitive load theory [35] also suggests that people learn better when information is split up into well organised sections.

Numbered sections, headings, and summary tables make a long answer easier to follow. This is why every template in our system asks for numbered sections in a fixed order.

H. Rubric-Based Evaluation

The text can be scored using human scoring, a model-based scorer [36] and rule based scoring. Human scoring is the most reliable option however is also the slowest and most expensive. Model based scorers offer flexibility however can be more expensive and may have bias. Rule based scoring is the cheapest fastest option which will always produce an identical answer. Each point can be easily linked back to an obvious reason. The downside to this is that they only look at surface level features. We have chosen to use a rule based scorer for our APEX scorer. This is because the tool needs to run in a very short amount of time (a fraction of a millisecond) and cannot access the network.

I. Model Context Protocol

MCP [16] is an open protocol which allows the integration between an AI host (e.g. chat assistant or code editor) and external servers. Messages are sent in JSON-RPC 2.0 format [37]. These messages can be requests, responses or notifications - all are small JSON objects. The server provides the model with 3 different functions: tools for the model to use, access to resources and choice of prompts for the user. A setup phase occurs at the beginning of a session where both parties advertise their capabilities. Tools can then be listed and called by the host. The easiest way to send messages is by using standard input/output. This is what our server uses.

III. Literature Review

A. Continuous and Discrete Prompt Search

Early works focused on finding prompts internally. Auto Prompt [38] uses gradient descent to find keywords to trigger the desired task. Prompt tuning [27] and prefix tuning [28] learn low dimensional vectors which can be appended to the input. While effective, these methods require access to the model weights and hence cannot be used for closed models. More recent works have investigated finding prompts in plain text. GrIPS [29] performs variations of the instruction by appending, deleting, or replacing phrases. Only variations which help are kept. RLPrompt [30] learns a small policy network using reinforcement learning to generate the prompt words.

B. Language Models as Prompt Optimizers

A newer line of work asks an LLM to improve prompts. Automatic Prompt Engineer (APE) [11] lets a model suggest many instructions from input and output examples, and then keeps the best one. ProTeGi [12] asks the model to describe the mistakes of the current prompt in words, which works like a text form of a gradient, and then changes the prompt using beam search. Yang et al. [10] introduced Optimization by PROmpting (OPRO). In OPRO, the model looks at earlier prompts and their scores, and then writes a new prompt. On GSM8K [39] and Big-Bench Hard [40], prompts found by OPRO did better than prompts written by people by up to 8% and 50%. EvoPrompt [41] combines LLMs with evolutionary algorithms, and Promptbreeder [42] even evolves the instructions that are used to change prompts.

Kepel and Valogianni [43] proposed Autonomous Prompt Engineering (APET). In APET, GPT-4 picks and uses methods such as expert prompting and chain of thought by itself. APET gave better results on tasks like word sorting and geometric shapes, but it struggled with harder strategy tasks, such as chess problems.

C. System Prompts and Data-Efficient Optimization

Most methods tune the user prompt for one task. Choi et al. [14] looked at the system prompt instead. Their MetaSPO method uses meta-learning with two loops: an inner loop that improves user prompts, and an outer loop that improves one system prompt shared by many tasks. The system prompts they found worked well on 14 new datasets from five areas. Wang et al. [13] proposed APEX, which saves cost by picking the most useful examples to score each new prompt. With a fixed number of calls, APEX made the starting prompt 11.2% better on average with one model. Chowdhury et al. [44] showed that a much simpler system, with no tuning and no hints about the task, can still do as well as complex methods at expanding short database column names.

D. Prompt Programming Frameworks

Some frameworks treat a chain of model calls as a program. In DSPy [15], the developer says what each step should do, and a compiler then finds good instructions and examples for each step. MIPRO [45] improves both the instructions and the examples in such multi-step programs. TextGrad [46] sends written feedback backward through a chain of calls, the same way gradients move through a neural network. In industry, the Prompt Generation framework [47] uses two JSON settings files to build the model input for a large online shopping search system. This shows that building prompts from clear settings is useful at a large scale.

E. Self-Refinement

Improvement does not always need a separate optimizer. Self-Refine [48] asks the same model to point out problems in its own answer and then rewrite it, and it does this a few times. Reflexion [49] saves short written lessons from failed tries and uses them in the next try. Our improvement stage uses the same idea of feedback and trying again, but the feedback comes from our rule-based scorer, not from a model.

F. Prompt Structure and Guidelines

Cook [50] developed PICCO, a framework reference containing five elements: (Persona), (Instructions), (Context), (Constraints) and (Output). This framework was developed from examining eleven existing prompting frameworks. Examples are not seen as a key element of the framework but rather a technique. This is simply a concept and was not investigated experimentally. Many prompting techniques have been identified and categorised according to their purpose in surveys by Schulhoff et al. [8] and Sahoo et al. [9]. The structure of our framework is similar to PICCO and contains 5 sections: (problem), (input), (context), (constraints) and (output format). The persona is decided in the separate strategy section.

G. Tool Use and Agents

A lot of recent work lets models take actions, not only write. ReAct [51] mixes reasoning steps with actions, such as search calls. Toolformer [52] showed that a model can learn by itself when to use tools such as a calculator. Gorilla [53] and ToolLLM [54] trained models to pick and call real APIs correctly. For several agents working together, AutoGen [55] and CAMEL [56] use conversation and role play. Agent Lightning [57] uses reinforcement learning to train agents to

plan and use tools better. These works show why a common tool protocol has become important.

H. Model Context Protocol Studies

Hou et al. [58] described the MCP ecosystem, the life of an MCP server, and the main security threats at each step. Hasan et al. [59] studied many open-source MCP servers and found security and upkeep problems. Yang et al. [60] reviewed protocols that let agents talk to tools and to each other, including MCP. None of these studies looked at prompt quality as a service given through MCP.

I. Prompt Security

Prompts can also be used to attack a system. Perez and Ribeiro [61] showed that a line like "ignore the previous instructions" can take control of a model. Greshake et al. [62] showed that harmful instructions can be hidden in web pages or emails that a model reads. This is called indirect prompt injection. AgentDojo [63] is a test setup for such attacks on agents. Chen et al. [64] used reinforcement learning to write injection text automatically, and their method worked better than earlier attacks on AgentDojo. Any tool that builds prompts from user text must keep these risks in mind.

J. Evaluation Benchmarks for Prompts

Prompt methods are usually tested on public benchmarks. GSM8K [39] has school-level math word problems, and each answer is a single number, so it is easy to check. Big-Bench Hard [40] is a set of 23 hard reasoning tasks where older models did worse than people. IFEval [65] checks if a model follows instructions that a program can verify, such as "write at least 300 words" or "do not use commas". In these benchmarks, each item has one clear right answer or one clear rule. Open questions, like "explain soft computing", do not have one right answer, so they are much harder to score automatically. This is one reason why our system scores the prompt itself with fixed rules, and why we also add a hand-checked case study.

K. Summary of the Review

Table I sums up the most relevant works. Table II compares our system with the main groups of methods on the features that matter for everyday use.

TABLE I. SUMMARY OF KEY RELATED WORKS

Work	Year	Method	Main Finding
APE [11]	2023	LLM proposes and ranks instructions	Model-written prompts can match human prompts
ProTeGi [12]	2023	Text feedback and beam search	Written error analysis guides prompt edits
OPRO [10]	2024	LLM as optimizer with scored history	Up to 50% better than human prompts on BBH
EvoPrompt [41]	2024	LLM with evolutionary algorithms	Evolution operators improve prompts steadily
APET [43]	2024	Self-chosen prompting techniques	Works on reasoning tasks, weak on strategy tasks
DSPy [15]	2024	Compiler for LLM pipelines	Less manual prompt writing in multi-step systems
MetaSPO [14]	2025	Meta-learning of system prompts	System prompts transfer to 14 unseen datasets
Survey [66]	2025	Review of heuristic search	Taxonomy of prompt search methods
No-cue APE [44]	2025	Simple meta-prompting	Simple design can match complex methods
PICCO [50]	2026	Synthesis of 11 frameworks	Five-part prompt structure, not yet tested
APEX [13]	2026	Data selection for evaluation	11.2% average gain under a fixed budget
MCP [16]	2025	JSON-RPC tool protocol	Standard way to link hosts and tools
AutoInject [64]	2026	RL-based prompt injection	Automatic attacks beat earlier baselines

TABLE II. COMPARISON OF THE PROPOSED SYSTEM WITH EXISTING APPROACHES

Feature	Soft prompt tuning	LLM-based optimizers	Prompt guides	Proposed system
Needs labeled data	Yes	Yes	No	No
Extra model calls	Training	Many	None	None
Works with closed models	No	Yes	Yes	Yes
Works for any new request	No	No	Yes	Yes
Asks the user questions	No	No	No	Yes
Explainable score	No	Partly	No	Yes
Available inside assistants	No	No	No	Yes (MCP)
Proven gain in answer quality	Yes	Yes	Partly	Not yet

L. Research Gap

Our review shows four gaps. First, most prompt optimizers need data, many model calls, or training, so they are hard for normal users to use. Second, structure frameworks such as PICCO are mostly ideas on paper and are rarely built into working tools. Third, research on MCP looks at the protocol and its security, not at the quality of the prompts that pass through it. Fourth, few tools explain why a prompt is good or bad. So there is a need for a light tool that works for any request without a dataset, explains its score, asks the user only a few simple questions, and runs inside the assistants people already use. The system in this paper tries to fill this gap. Table III shows how its parts build on earlier work.

TABLE III. MAPPING OF PRIOR WORK TO THE MODULES OF THE PROPOSED SYSTEM

Work	Main Idea	Use in Our System
APET [43]	Model selects prompt techniques by itself	Stage 5: action, persona, execution, and target
PICCO [50]	Five-part prompt structure	Stage 6: structured prompt blueprint
MetaSPO [14]	Improves the shared system prompt	Stage 7: system role, policy rules, and guardrails
APEX [13]	Scores prompts before accepting them	Stage 9: four-dimension quality gate
OPRO [10]	Improves prompts using scored history	Stage 10: refinement driven by the list of flaws
MCP [16]	Standard tool protocol for assistants	Server interface with 17 tools

IV. PROPOSED SYSTEM

A. Design Goals

We set five design goals. The system must accept any request without training data. It must give a result in much less than one second. It must ask the user as little as possible. Every score must be easy to explain. Lastly, it must run both on its own and inside an MCP host.

B. System Architecture

The architecture of the system is shown in Fig. 1. The client layer provides two ways to interact with the system, either through a terminal or with an MCP host such as Claude Desktop, Claude Code, or Cursor. The transport layer handles JSON-RPC 2.0 messages via standard input and output. The server core contains the tool interface, argument parser and the task classifier, question module, prompt builder, APEX evaluator and refiner. The storage layer contains a JSON file for every pipeline session.

C. Overall Workflow

Fig. 2 outlines a prompt design process. Raw requests are analyzed for intent, extracting context, domain classification, selecting strategies, designing prompts and prompting the system. The prompt is evaluated using the APEX evaluator and if it does not score above the pass mark then it is refined and re-evaluated. This continues until it passes or reaches three iterations. Table IV shows the breakdown of the 11 different stages and the tools used.

There are two options available within the system. The fast option only requires two tool calls. The first returns the questions and the second returns the prompts and score. Within the second call stages 2-9 are performed as usual function calls. The granular option has each stage as an individual tool meaning each can be controlled by the external agent, storing their own variables.

.

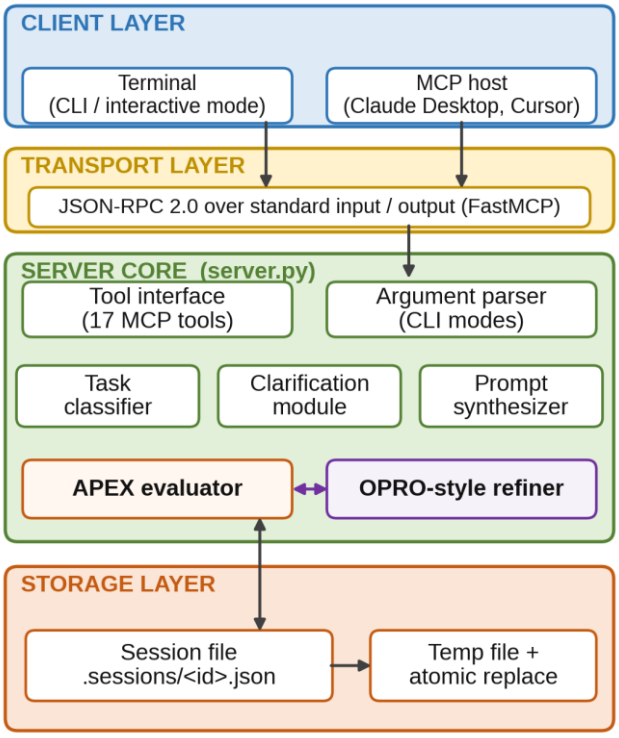


Fig. 1. Layered architecture of the Adaptive Universal Prompt Optimizer.

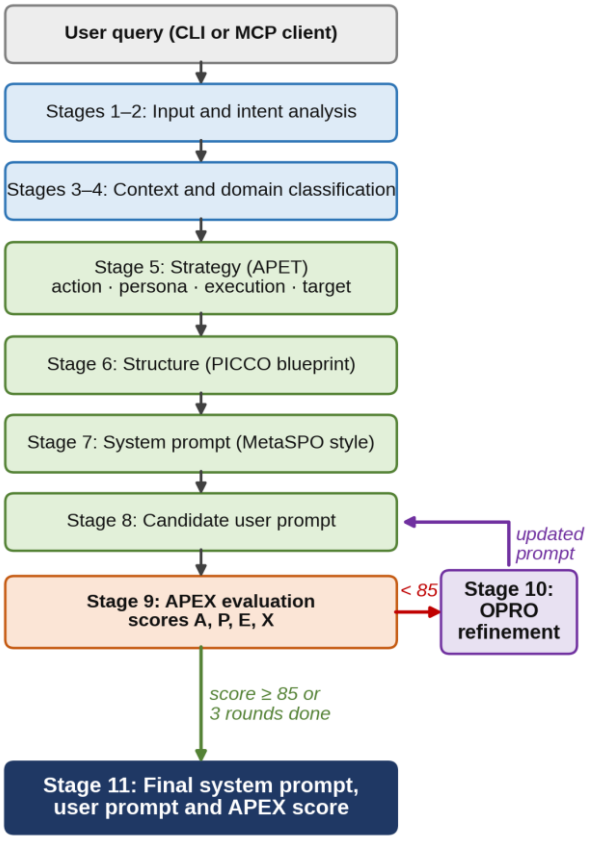


Fig. 2. The 11-stage workflow. Blue stages analyze the request, green stages build the prompt, orange scores it, and purple refines it.

TABLE IV. THE 11 PIPELINE STAGES AND THEIR TOOLS

No.	Stage	MCP Tool and Stored Data
1	Input	start_pipeline: raw request, new session ID
2	Intent	submit_intent_analysis: goal, task type, keywords
3	Context	submit_context_extraction: audience, tools, constraints, features
4	Domain	submit_domain_classification: domain and confidence
5	Strategy (APET)	submit_strategy_apet: action, persona, execution, target
6	Structure (PICCO)	submit_structure_picco: problem, input, context, constraints, output format
7	System prompt	submit_system_prompt_metaspo: role, policy rules, output steering, guardrails
8	Candidate	submit_candidate_prompt: draft user prompt and its versions
9	APEX evaluation	submit_apex_evaluation: scores, flaws, pass or fail
10	Refinement	submit_opro_refinement: refined prompt and changes made
11	Final	finalize_prompt: final prompts, score, full trail

D. Task Classification

The classifier first changes the request to lower case. Then it checks six ordered groups of keywords, as shown in Fig. 3. The first group that matches decides the task type, the expert persona, the structure template, and the quality focus. If no group matches, the request goes to the general "expert consultation" type. Table V lists the six types.

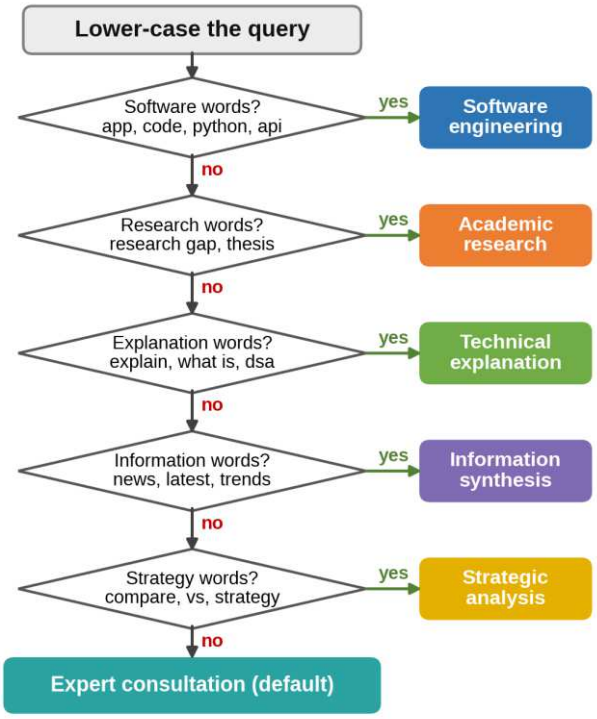


Fig. 3. Decision flow of the ordered keyword classifier. The first matching group decides the task type. Each task type keeps the same colour in all later figures.

TABLE V. TASK TYPES, EXAMPLE TRIGGER WORDS, AND PERSONAS

Task Type	Example Trigger Words	Assigned Persona
Software engineering	app, code, python, api, backend	Principal Software Architect and Full-Stack Product Engineer
Academic research	research gap, literature review, thesis	Senior Academic Researcher and Literature Methodologist
Technical explanation	explain, what is, algorithm, dsa	Master Computer Science and STEM Educator (Feynman Technique)
Information synthesis	news, latest, recent, trends	Senior Technology Intelligence Analyst and Research Specialist
Strategic analysis	compare, vs, strategy, pros and cons	Principal Strategic Consultant and Decision Analyst
Expert consultation	no match (default)	World-Class Domain Specialist and Rigorous Analytical Consultant

The order of the groups is important. For example, the request "explain research gap in llms" has both an explanation word and a research phrase. Since the research group is checked first, it is treated as a research task, which is the better choice. The matching works on parts of words. This is simple, but it can cause mistakes, as we show in Section VII.

Algorithm 2: Ordered keyword classification

```
Input : raw request q
Output: task type, persona, template
1: t <- lower case(trim(q))
2: for each group G in [software, research,
       explanation, information, strategy] do
3:   for each keyword k in G.keywords do
4:     if k is a substring of t then
5:       return (G.type, G.persona, G.template)
6: return (consultation, default persona,
          general template)
```

E. Adaptive Clarification

Before the final prompt is built, the system asks three short questions, shown in Table VI. Every question has a default, so the user can reply with one word, "default". If the user says "quick", "fast", or "skip questions", this step is skipped. The answers are turned into short lines, called directives, which are added to the end of the user prompt under the heading "User-Specified Refinements".

TABLE VI. CLARIFICATION QUESTIONS, OPTIONS, AND DEFAULTS

Question	Options	Default
Q1 Depth	1: quick overview; 2: balanced practical guide; 3: deep dive with edge cases	2 (balanced)
Q2 Elements	1: definitions and analogies; 2: real examples; 3: step-by-step guidance; 4: trade-offs and pitfalls; 5: all	5 (all)
Q3 Audience	Free text, for example "beginner Python developer"	General

F. Prompt Synthesis

Each task type has its own template. The system prompt names the persona and says how it works. For example, the research persona is told to give careful answers based on evidence and to keep real data apart from guesses. The user prompt repeats the original request and then adds four or five numbered sections. Fig. 4 shows the parts of an improved prompt for a software request, and Table VII lists the sections used by each template.

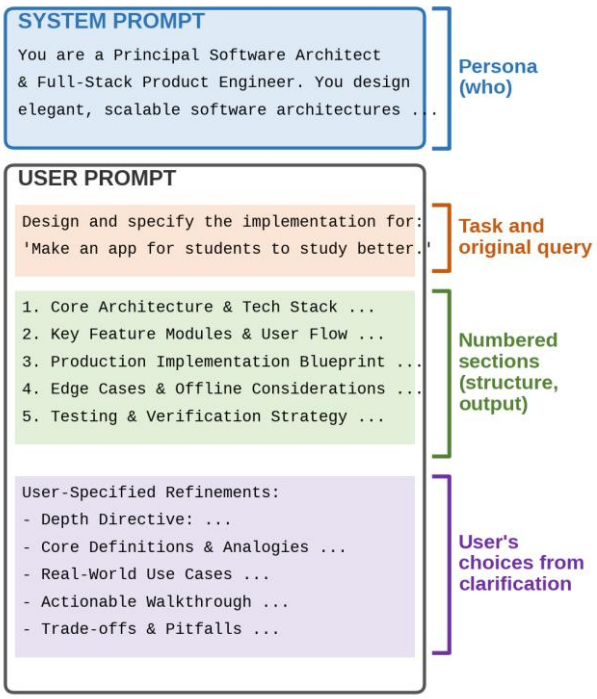


Fig. 4. Parts of an optimized prompt for the request "Make an app for students to study better".

TABLE VII. NUMBERED SECTIONS USED BY EACH TEMPLATE

Template	Sections Requested in the User Prompt
Software	Architecture and tech stack; feature modules and user flow; implementation blueprint; edge cases and offline use; testing and verification
Research	Summary and problem framing; root causes and findings; research gaps and limits; hypotheses and future studies; measurement and verification
Explanation	Everyday analogy; core ideas and first principles; walkthrough and trade-offs; misconceptions and pitfalls; practice problem
Information	Recent developments; technical drivers; impact on industry and society; limits and open challenges; sources and key metrics
Strategy	Context and objectives; comparison of options; trade-offs and hidden costs; recommendation
General	Objective breakdown; structured answer; constraints and edge cases; action summary

G. Worked Example

To make the process clear, we follow one real request, "explain research gap in llms", through the system. The raw request has only five words. When it is scored alone, it gets 36.0 points: 52 for Assessment and the lowest possible value of about 30 for the other three parts. The scorer lists nine problems, such as no goal, no persona, no numbered steps, and no guardrails.

In the classification stage, the phrase "research gap" matches the research group. So the request gets the research template and the persona "Senior Academic Researcher and Literature Methodologist". The user accepts the default answers to the three questions. The builder then writes the system prompt and the user prompt shown in Box 1.

Box 1: Optimized prompt for "explain research gap in llms"

```
SYSTEM: You are a Senior Academic Researcher &
Literature Methodologist. You provide exceptionally
rigorous, evidence-backed academic synthesis
distinguishing empirical data from speculation.
Conduct an academic research inquiry into: 'explain
research gap in llms'
Structure your analysis in these distinct sections:
1. Executive Summary & Problem Framing: Define the
theoretical domain and key foundational studies.
2. Root Cause Analysis & Empirical Findings: Synthesize
peer-reviewed findings and documented causal factors.
3. Identified Research Gaps & Methodological
Limitations: Highlight under-explored factors and data
collection biases in existing studies.
4. Actionable Hypotheses & Future Study Directions:
Propose empirical frameworks and testable hypotheses to
investigate these gaps.
5. Measurement & Verification Framework: Detail metrics
and evaluation criteria to validate these findings.
User-Specified Refinements:
- Depth Directive: Provide a balanced, practical
explanation with step-by-step clarity and actionable
depth.
- Core Definitions & Intuitive Analogies: Ground
foundational concepts in relatable real-world analogies
and clear definitions.
- Real-World Use Cases: Include practical, concrete
examples demonstrating the concepts in realistic
scenarios.
- Actionable Walkthrough: Detail step-by-step
procedures, architectural blueprints, or implementation
guidelines.
- Trade-offs & Pitfalls: Explicitly analyze critical
trade-offs, potential failure modes, common
misconceptions, and edge cases.
```

This prompt received an overall score of 75.5. This was calculated by adding the score from each section together (80 from assessment, 60 from performance, 80 from execution and 82 from excellence). The only issue with this prompt was that there was no identified audience. If an audience had been provided the score would have been increased to 80.5. This shows how a short phrase can be developed into a clear structure with 5 sections. The user's original wording has also been used.

Another example showing how the answers to the questions affect the prompt. For the request 'What is cloud computing', the user selected the brief summary, only wanted real examples and identified the audience as school students. This meant the prompt asked for a brief answer using real examples. It also asked to use simple words for school students. This prompt received an overall score of 68.2. This also shows one of the weaknesses of the templates. The explanation template was originally created for data structures, meaning it asks about time and space complexity. This doesn't make sense when talking about something like cloud computing.

H. APEX Quality Evaluation

The APEX evaluator scores a prompt on four parts, as shown in Fig. 5. Each part starts from a low base value and gains points step by step, based on how many matching items it finds. Table VIII gives the full rules. The Assessment part also looks at how many important words from the raw request are kept in the final prompt. Let Q be the set of words in the request that are longer than three letters and are not common filler words, and let U be the user prompt. The keyword retention ratio is:

$$r = |\{ w \in Q : w \text{ appears in } U \}| / |Q| \qquad (2)$$

and the Assessment dimension gains $22r$ points from it. The score of each dimension is then limited to a fixed range:

$$S_d = \min(\, 93, \max(\, 30, b_d + g_d \,) \,) \tag{3}$$

where b_d is the base value of dimension d and g_d is the sum of its bonus points. The overall score is the mean of the four dimensions:

$$\text{APEX} = (\, S_A + S_P + S_E + S_X \,) / 4 \tag{4}$$

If the prompt scores 85 or above on the APEX scale then it would pass. As the maximum score in each section is 93 the prompt needs to perform well across all areas. The feedback includes the score and a list of issues. For example a prompt might have no identified audience and/or not ask the model to handle edge cases.

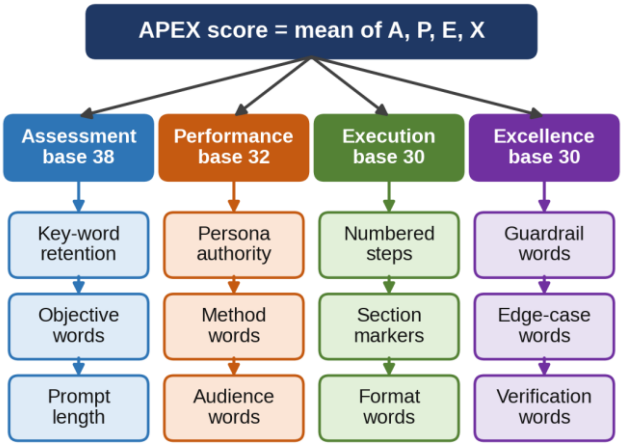


Fig. 5. Structure of the APEX score. Each dimension has a base value and three groups of criteria.

TABLE VIII. APEX SCORING RUBRIC

Dimension	Criterion	Points
Assessment (base 38)	Retention of request key words	Up to 22 (22 × r); 12 if no key words
	Objective words (task, goal, scope ...)	2 or more: 14; one: 8
	Length of user prompt	80 to 280 words: 12; near range: 6; under 50: minus 8
Performance (base 32)	Authority words in persona	3 or more: 22; two: 16; one: 10
	Method words (rigorous, evidence ...)	3 or more: 18; two: 12; one: 6
	Audience words (student, developer ...)	2 or more: 14; one: 8
Execution (base 30)	Numbered steps	5 or more: 24; 3 or 4: 16; 1 or 2: 8
	Section markers (**, ##, requirements:)	3 or more: 18; 1 or 2: 10
	Format words (table, summary, outline ...)	3 or more: 16; 1 or 2: 9
Excellence (base 30)	Guardrail words (avoid, do not ...)	4 or more: 22; 2 or 3: 14; one: 7
	Edge-case words (pitfall, risk ...)	3 or more: 22; two: 14; one: 7
	Verification words (test, validate ...)	3 or more: 16; 1 or 2: 9

I. OPRO-Style Refinement

When a prompt fails, the improvement stage reads the list of problems. Each type of problem is linked to a new requirement, as shown in Table IX. The new lines are added under the heading “Mandatory Refinement Requirements”. If no rule matches a problem, a general quality line is added instead. The host agent can also send its own improved prompt, which is accepted if it has at least 30 characters. The new prompt is then scored again. Algorithm 1 shows the loop. It follows the main idea of OPRO [10]: score, learn from the score, and try again. But it uses rules instead of a model, so it needs no extra model calls.

TABLE IX. RULES FOR TURNING FLAWS INTO REQUIREMENTS

Flaw Contains	Requirement Added to the Prompt
“analogy”	Anchor the main idea in a familiar real-world analogy
“misconception”	Point out and correct common misunderstandings
“gap” or “research”	Name 3 or 4 open problems and current method limits
“edge case”	Describe failure modes and boundary conditions
“format” or “structure”	Organize the output with clear, ordered headings
No rule matches	Ensure factual accuracy, clarity, and complete structure

Algorithm 1: APEX-guided refinement loop

```
Input : request q, system prompt s, user prompt u
Output: final prompt u*, score a*
1: k <- 0
2: repeat
3:    (a, F) <- APEX(q, s, u)    // score, flaws
4:    if a >= 85 or k = 3 then return (u, a)
5:    L <- empty list
6:    for each flaw f in F do
7:      if a rule matches f then add its line to L
8:    if L is empty then add quality line to L
9:    u <- u + heading + L
10:   k <- k + 1
11: until done
```

J. MCP Interface

The server offers 17 tools, listed in Table X. The fast-path tools are for normal use. The pipeline tools are for programs that want full control. When it starts, the server also sends a short set of instructions to the host. These tell the host to ask the questions first, to use at most two tool calls for one optimization, to copy the result without changes, and to always show the full APEX score. Fig. 6 shows the messages sent on the fast path.

TABLE X. TOOLS EXPOSED BY THE MCP SERVER

Tool	Purpose
get_adaptive_clarification_ questions	Returns the three questions for a request
optimize_prompt	Builds and scores the final prompts using the user’s answers
quick_optimize_prompt	Builds and scores prompts without asking questions
optimize_explanation_ prompt	Older name kept for explanation requests
start_pipeline ... finalize_ prompt	Eleven stage tools listed in Table IV
get_pipeline_status	Returns the current stage and progress, such as 5/11
list_sessions	Lists all saved sessions

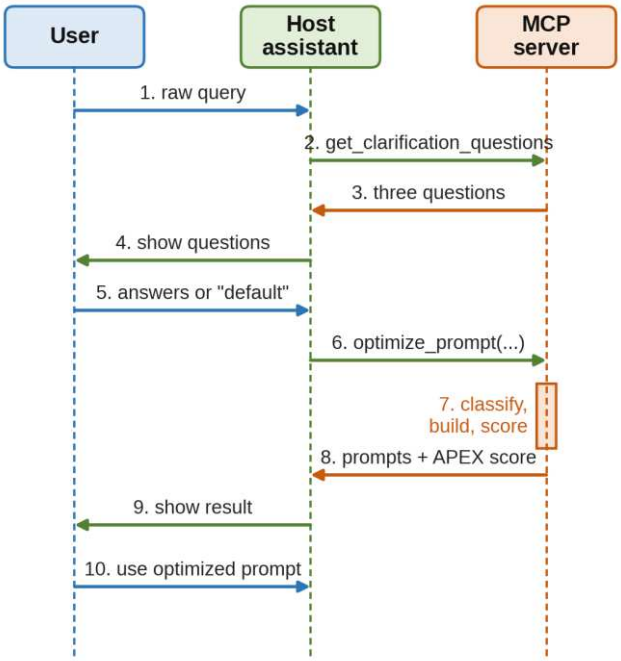


Fig. 6. Message sequence on the fast path. Only two tool calls are needed for one optimization.

K. Message Format

Every tool call is a JSON-RPC request with the method name "tools/call". The request gives the tool name and its arguments, and the server sends back a result with the same ID. Box 2 shows a short example of the second call on the fast path. Since the whole exchange is plain JSON text, it is easy to log, test, and debug.

Box 2: Shortened JSON-RPC exchange for optimize_prompt

```
--> {"jsonrpc": "2.0", "id": 7,
     "method": "tools/call",
     "params": {"name": "optimize prompt",
       "arguments": {"user query": "tell me about DSA",
         "depth": "2", "elements": ["5"],
         "target audience": "beginner",
         "skip clarification": true}}}
<-- {"jsonrpc": "2.0", "id": 7,
     "result": {"status": "success",
       "detected task":
         "technical conceptual explanation",
       "final system prompt": "You are a Master ...",
       "final user prompt": "Provide a ...",
       "apex quality score": ...,
       "apex breakdown": {...}}}
```

L. Session Management and Operation Modes

Each session is stored in a JSON file with a unique 12 character ID. The JSON contains the original request, current stage, the number of optimization steps, all data of the stages and the history with timestamps. Each stage has a stage guard that makes sure the tools are used in the correct order. The only exception is the scoring and optimization loop. There are three different terminal modes (besides the MCP server). The request can be given in the terminal by typing it in or using the -q option. When using -i the program will ask the three questions interactively. With --test you can do a small test with 4 requests.

V. Implementation

This system was written in Python and can therefore be contained within a single 1170 line file. It requires Python version 3.10 or later and the mcp package to give the FastMCP package. The main functions are listed in table XI. All session files will be stored safely by using a temporary file. The file will be written into and then atomically swapped with the old file. This means that if the program crashes the file will not be in an incomplete state. The MCP server can easily be added to Claude Desktop by adding a short JSON file containing the path to the Python program and the path to the server file. The project is also available under the MIT license.

TABLE XI. Main Functions in the Source Code

Function	Role
classify_prompt_dynamically	Finds the task type, persona, and template
generate_adaptive_clarification_questions	Builds the three questions
synthesize_optimized_prompt	Builds the system and user prompts and adds directives
evaluate_prompt_universally	Computes the four APEX scores and the flaws
Session.save and _load	Write and read session files
_require_stage	Checks the order of the pipeline stages
run_interactive_cli	Runs the question-and-answer mode in a terminal
run_self_test	Runs the built-in check

While testing, we found one bug in the code. The scoring tool of the granular path calls the scoring function with only two arguments, but the function needs at least three. Because of this, the granular path stops with an error at stage 9. The fast path is not affected. So, for the improvement test in Section VII, we ran the same improvement rules directly and called the scoring function with the request, the system prompt, and the improved user prompt.

A. Cost Analysis

The system incurs low and predictable costs. To calculate the costs of the system, we consider a request of n characters, K trigger keywords in total, and an output prompt of m characters. Since each keyword is checked once against the request by the classifier, the cost of the classifier is:

$$C_{cls} = O(K \cdot n) \quad (5)$$

The scorer checks a fixed list of W scoring words against the prompt and counts the numbered lines, so its cost is:

$$C_{eval} = O(W \cdot m) \quad (6)$$

K is approximately 50, W is approximately 80, and m is approximately 1,300 in our system, so both stages are very quick. The refinement process will be performed a maximum of 3 times. Therefore a maximum of 3 extra evaluations will be made. None of these stages involve using a language model, meaning they do not involve using a network or GPU. The memory requirements are low as we only store the request, two prompts and scores. This is in contrast to methods such as OPRO [10] or APE [11], which require hundreds or thousands of model calls for a single task. This means that our tool can run easily on a normal laptop and provides instant results.

VI. Experimental Setup

A. Test Set

We wrote a test set of 60 requests, with ten requests for each of the six task types. The requests are short and informal, like the ones students type every day. Twelve of them come from the project notes and from the case study described below. Table XII gives examples. For each request, we wrote down the task type that a careful person would pick. We call this the intended type.

TABLE XII. Test Set Summary

Intended Type	Count	Example Requests
Software	10	Write a Python script to rename files; build a website for a small bakery
Research	10	Find the research gap in blockchain for healthcare records
Explanation	10	What is a binary search tree; explain recursion to a beginner
Information	10	Latest trends in electric vehicles; updates on space missions
Strategy	10	Compare SQL vs NoSQL databases for a startup; business plan for a coffee shop
Consultation	10	How to prepare for campus placements; tips to improve public speaking

B. Test Conditions

Each request was scored in three conditions:

- *Raw:* the request itself was scored as the user prompt, with an empty system prompt.
- *Optimized:* the request was improved with the default answers (balanced depth, all elements, general audience).
- *Optimized with audience:* the request was optimized with the deep-dive setting and the audience "undergraduate engineering students".

We also ran four more tests. In the first, we tried all 15 pairs of depth and element choices. In the second, we tried three different audiences. In the third, we ran the improvement loop on all 60 improved prompts. In the fourth, we measured running time. Lastly, we tested five extra requests that have trigger letters inside longer words.

C. Metrics

For classification, we used accuracy, precision, recall, and the F1 score. Let TP, FP, and FN be the true positives, false positives, and false negatives for one task type, and let N be the number of requests:

$$\text{Accuracy} = (\text{number of correct labels}) / N \quad (7)$$

$$\text{Precision} = TP / (TP + FP) \quad (8)$$

$$\text{Recall} = TP / (TP + FN) \quad (9)$$

$$F1 = 2 \times \text{Precision} \times \text{Recall} / (\text{Precision} + \text{Recall}) \quad (10)$$

For prompt quality, we used the APEX score from (4). We report the mean, the standard deviation, and the relative gain:

$$\text{Gain}\ (\%) = 100 \times (\text{mean}_{opt} - \text{mean}_{raw}) / \text{mean}_{raw} \quad (11)$$

We also report the pass rate, which is the share of prompts that score at or above a given threshold.

D. Case Study Design

The APEX scorer uses the same rules that shape the templates, so a higher score is partly expected by design. To check the effect on real answers, the project team ran a case study with eight requests. Each request was given to a general chatbot twice, once as typed and once in its improved form, and the two answers were compared by hand.

E. Environment

All tests ran offline on a Linux machine with an Intel Xeon processor at 2.10 GHz and Python 3.12.3. The tested functions do not use the network. The mcp package was not installed on the test machine, so we replaced its import with a small stand-in. This does not change the logic of the tested functions.

VII. Results and Discussion

A. Overall Score Change

Table XIII gives the overall results. The mean APEX score went up from 36.4 for raw requests to 70.9 for improved prompts. This is a gain of 34.5 points, or about 95%. With the deep-dive setting and an audience, the mean went up to 74.5. Raw requests had very close scores, from 33.5 to 38.2, because none of them have structure, a persona, or guardrails. Improved scores were more spread out, from 62.2 to 77.5, because some templates are stronger than others. Fig. 7 shows how the scores are spread.

TABLE XIII. Overall APEX Results on 60 Requests

Measure	Raw	Optimized	Opt. + audience
Mean score	36.4	70.9	74.5
Standard deviation	0.9	4.3	4.2
Lowest score	33.5	62.2	67.2
Highest score	38.2	77.5	80.5
Gain over raw	–	+34.5 (95%)	+38.1 (105%)
Mean flaws per prompt	9.8	1.6	–
Prompts that pass (85)	0	0	0

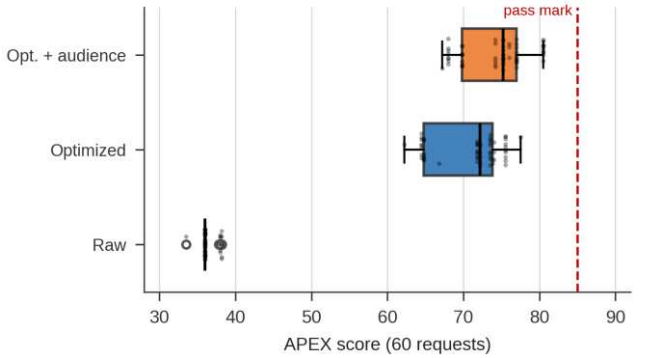


Fig. 7. Spread of APEX scores for the 60 requests in the three conditions. The dashed line is the pass mark.

B. Results by Dimension

Overall average scores are presented in Figure 8. The highest improvements were found within Execution and Excellence. Execution improved from 30.4 to 74.3 and Excellence improved from 30.6 to 72.0. This is because these are scored based on numbered parts, bullet points and guardrail use. These will rarely be found in a raw request. Assessment improved from 52.0 to 81.7. Performance had the lowest improvement, increasing from 32.7 to 55.4. However,

this did improve to 66.6 when an audience was given. The only components that were affected by the deep dive configuration and giving an audience were Assessment and Performance as there were no new numbered steps or guardrail language within the new lines.



Fig. 8. Mean APEX score of each dimension for raw requests and optimized prompts (60 requests).

C. Results by Task Type

Fig. 9 and Table XIV show the results for each intended task type. Research requests gained the most and reached 80.5 with an audience. Explanation and strategy requests gained the least. Table XV shows why. It gives the mean of each part for each template. The explanation and strategy personas have fewer authority and method words, so their Performance scores were only about 49 and 42. The strategy template also has only four numbered sections, which lowers its Execution score.

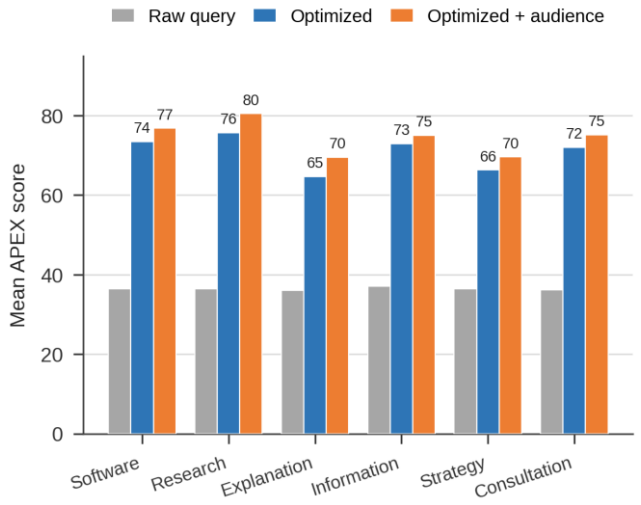


Fig. 9. Mean APEX score for each intended task type.

TABLE XIV. MEAN APEX SCORE BY INTENDED TASK TYPE

Task Type	Raw	Optimized	Opt. + audience
Software	36.4	73.5	76.8
Research	36.4	75.7	80.5
Explanation	36.0	64.7	69.5
Information	37.1	72.9	75.0
Strategy	36.4	66.3	69.6
Consultation	36.2	72.0	75.2

TABLE XV. MEAN DIMENSION SCORES OF EACH TEMPLATE (OPTIMIZED)

Template	A	P	E	X
Software	86.0	60.7	80.0	68.0
Research	80.0	60.8	80.0	82.0
Explanation	79.0	48.8	70.0	61.0
Information	72.0	62.0	80.0	77.3
Strategy	86.0	42.0	62.0	68.0
Consultation	86.0	55.3	72.0	75.0

D. Classification Accuracy

From a total of 60 requests, the classifier correctly classified 56 of them, giving an overall accuracy of 93.3%. This can be seen in the confusion matrix within Fig 10 and in the precision, recall and F1 values found in Table XVI. Research and explain requests were always correct. The average F1 over the 6 request types was 0.94.

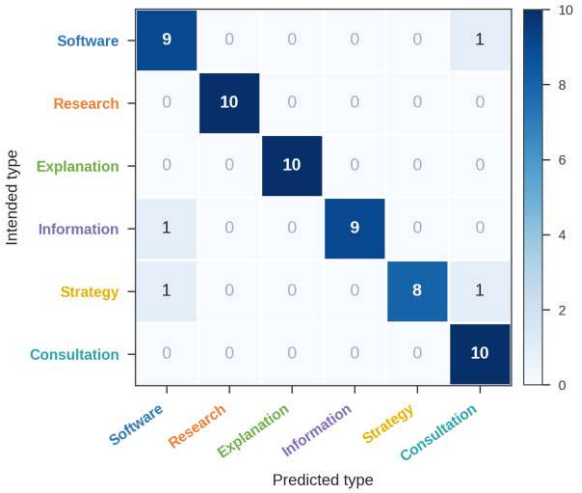


Fig. 10. Confusion matrix of the keyword classifier on 60 requests.

TABLE XVI. PRECISION, RECALL, AND F1 SCORE OF THE CLASSIFIER

Task Type	Precision	Recall	F1
Software	0.82	0.90	0.86
Research	1.00	1.00	1.00
Explanation	1.00	1.00	1.00
Information	1.00	0.90	0.95
Strategy	1.00	0.80	0.89
Consultation	0.83	1.00	0.91

Table XVII lists all mistakes. Two mistakes happened because a trigger word showed up in a request of another type: “python” in a news request and “software” in a strategy request. Two mistakes happened because there was no trigger word at all: “chatbot” is not in the software list, and “should I choose” is not in the strategy list. The five extra requests showed a second problem. Because the classifier matches parts of words, “approach”, “apply”, and “happy” all contain the trigger “app”. So three of the five requests were wrongly sent to the software template.

TABLE XVII. CLASSIFICATION ERRORS

Request	Intended	Predicted
Make a chatbot for college admission queries	Software	Consultation
Find information about new Python releases	Information	Software
Pros and cons of remote work for a software company	Strategy	Software
Should I choose MBA or MTech after engineering	Strategy	Consultation
What is the best approach to learn guitar*	Explanation	Software
Explain what happens when I apply for a passport*	Explanation	Software
Tell me about happy hormones*	Consultation	Software

[a] Extra requests used only to test matching inside longer words.

E. Effect of the User's Choices

Fig. 11 shows the mean score for all 15 pairs of depth and element choices. The quick and balanced depths gave the exact same scores, because their lines have no scored words. The deep-dive setting added about one point, from 70.9 to 71.6 with all elements, because its line mentions edge cases. Picking all elements gave the highest scores, and picking only examples gave the lowest. In total, these choices changed the score by less than three points. Their main value is in what the answer contains, not in the score.

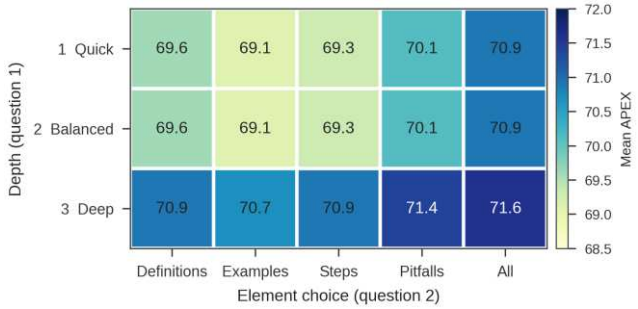


Fig. 11. Mean APEX score for each pair of depth and element choices (general audience).

The audience had a bigger effect. Any audience raised Performance from 55.4 to 66.6 and the total score from 70.9 to 73.6. But Table XVIII shows that three very different audiences got the exact same score. The scorer only checks that an audience line is there. It cannot tell if the audience makes sense for the request.

TABLE XVIII. EFFECT OF THE AUDIENCE ON THE APEX DIMENSIONS

Audience	A	P	E	X	APEX
None (general)	81.7	55.4	74.3	72.0	70.9
Engineering students	81.7	66.6	74.3	72.0	73.6
Beginner Python developer	81.7	66.6	74.3	72.0	73.6
Non-technical manager	81.7	66.6	74.3	72.0	73.6

F. Refinement Loop

The mean score on Fig. 12 shows that there was an increase after the first round, rising to 74.9 compared to 70.9. However, this was because 48 out of the 60 prompts had improved. After the first round there was no change in the score. Looking at Table XIX we can see why the score didn't change. This table shows the issues which were identified by the scorer in the refined prompts prior to entering the improvement loop. However, none of the problems identified can be found in any of the rules in Table IX. This meant that every time the refiner was used, the general quality line was always added. This meant that the score increased slightly due to new words being added to this line and also because of the title. However, in further rounds of the improvement loop the same line was added. As each word is only counted once during scoring, the score remained the same. No prompts passed.

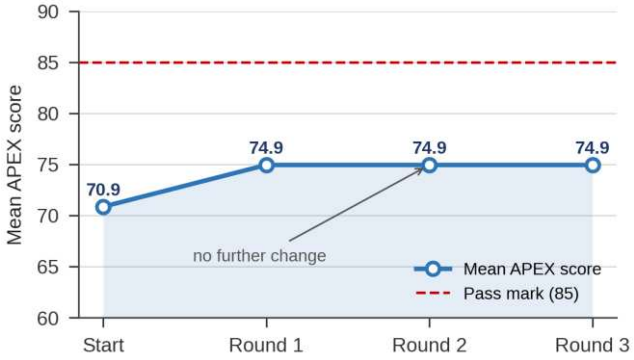


Fig. 12. Mean APEX score over three rounds of refinement (60 prompts).

TABLE XIX. FLAWS REPORTED FOR OPTIMIZED PROMPTS BEFORE REFINEMENT

Flaw	Prompts
Performance: no clear audience	46
Excellence: no anti-hallucination rule	21
Performance: no method or quality standard	17
Assessment: no clear objective statement	9

This result shows a gap between the scorer and the refiner. The scorer can name a problem, but the refiner has no rule to fix it. Adding rules for audience, guardrails, and goals would close this gap.

G. Choice of Pass Mark

Fig. 13 shows how many prompts pass at each possible threshold. No raw request passes any threshold above 40. At a threshold of 70, 70% of improved prompts pass. At 75, 18% of improved prompts and 60% of prompts with an audience pass. At 80, only 17% of prompts with an audience pass, and at 85 none pass. A threshold between 70 and 75 would clearly separate raw requests from improved prompts. The current value of 85 is too strict for the current templates.

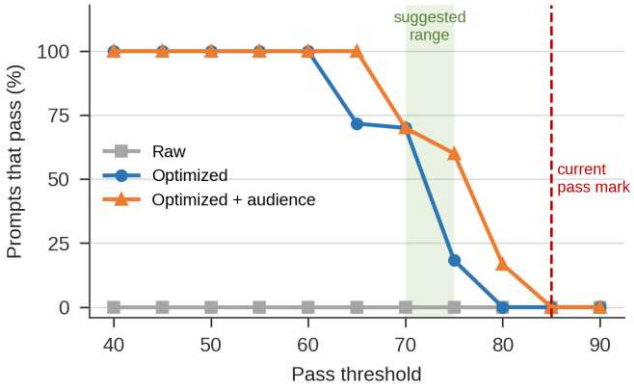


Fig. 13. Share of prompts that pass for different threshold values.

H. Prompt Length and Running Time

Fig. 14 shows the number of words. Raw requests had 7.2 words on average. Improved prompts, counting both the

system prompt and the user prompt, had 180.7 words, from 149 to 198. The user prompt alone stayed inside the range of 80 to 280 words that the scorer likes. Longer prompts use more tokens, but about 180 words is small for modern models.

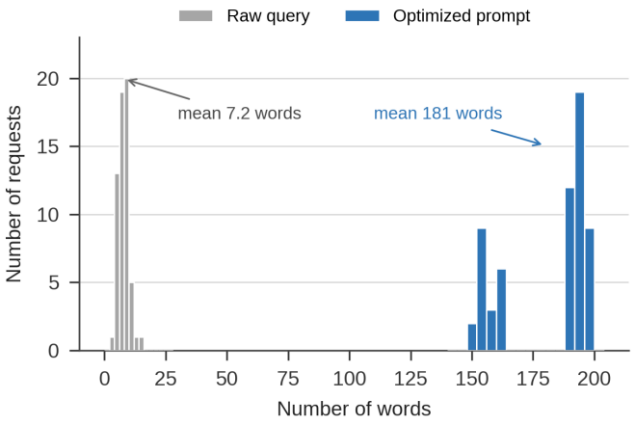


Fig. 14. Number of words in raw requests and in optimized prompts.

Table XX shows the running time. Building and scoring one prompt took less than 0.1 ms on average. The classifier alone took about 3 microseconds. So the system adds no delay that a user could notice. In real use, the total time depends on the host assistant and the model, not on our tool.

TABLE XX. MEAN RUNNING TIME PER REQUEST

Operation	Time (ms)
Task classification	0.0029
APEX evaluation	0.091
Full synthesis (classify, build, score)	0.095

I. *Case Study of Real Answers*

As shown in Table XXI, all of the eight cases were improved by using the enhanced prompts to provide clarity in the structure for the chatbot. In the health care case, the original answer was very brief, listing only a few areas. However, the enhanced prompt provided a structure asking for an introduction, 6 areas of health care with real examples, benefits, challenges, ethics and regulations, and a future outlook of 800-1200 words. This was adhered to, along with keeping existing tools in the answer but omitting any tools still being researched. In the crop disease case, the enhanced prompt asked for tables to be created showing datasets and methods, and the answer contained these along with a clear distinction between lab and field results.

TABLE XXI. CASE STUDY: ANSWERS TO RAW AND OPTIMIZED PROMPTS

Request	Raw Prompt Answer	Optimized Prompt Answer
AI in health care	List of seven areas and short challenges	Six planned sections, examples, benefit table, ethics, outlook
ML for crop disease review	Long review with many accuracy figures	Abstract, method taxonomy, dataset and method tables, lab versus field gap
Research gap in LLMs	List of twelve gaps with a table	Four focused gaps with study ideas and ways to measure progress
Soft computing	Overview of four pillars	Seven sections, an analogy for each pillar, comparison table
Social media and students	Long list of ideas in six themes	Short answer with assumptions, pitfalls, and next steps
Explainable AI and trust	Methods and eight gaps	Types of trust, method table, gaps written as research questions
“I need a research topic”	General topics with no context	Thirteen topics fitted to the student’s field and level
“Study machine learning”	Short concept map	Nine-part study guide with pitfalls and a learning plan

The case study also showed some costs. Improved answers were much longer, which takes more time to read and uses more tokens. For a simple question, this extra length may not be needed. The depth question helps here, because the quick overview setting asks the model to keep the answer short, as seen in the social media case. In one case, an older version of the scorer gave the improved prompt only 64 points, but the answer was still clearly better planned than the raw one. This again shows that the APEX score and the real quality of the answer are linked but not the same.

J. *Discussion*

The results show that there are three key findings. First, the basic rule-based system can generate a clear and complete prompt from an ambiguous prompt quickly and for a variety of different prompts. Second, the ordered keyword classifier performs well on standard student prompts but fails when presented with novel phrasing and trigger letters in longer words. Third, the scoring and refinement systems help to explain prompts but have not been optimised yet for one another or actual answer quality.

This matches previous work. Guidelines [26] and [25] recommend specifying who the audience is, what the structure will be and any constraints. This is exactly what contributed to our increased scores and determined how the answers were produced for the case study. However, there is also previous work which shows that there are aspects of prompts which cannot be captured using a set of rules [5] [6]. Therefore whilst this provides a good indication of the structure of the prompt, we should not consider it to guarantee a better answer will be produced based on the APEX score.

K. *Error Analysis of the Templates*

Looking at the improved prompts we can identify three issues with the templates. Firstly there is a content issue. The explanation template was originally designed for data structures and algorithms so will always ask for the time and space complexity. It will also always ask for a practice problem with the best solution and a slower solution. This issue is present in the ‘What is a binary search tree’ prompt

but not in the ‘How does the internet work’ or ‘What is cloud computing’ prompts. Secondly there is a lack of persona in the strategy template. There is only one word of authority and no methodology word, therefore every strategy prompt will lose marks in Performance. Lastly, there is a lack of specificity in the general template. It will always ask for an objective breakdown and structured deliverable. These are correct words but very vague meaning the response will vary depending on the model. If the prompt was something like “Give me a diet plan for exam time” it would be better to have a personal advice template with clear safety notes.

Whilst these issues will not prevent the tool functioning they do show that every template should be tested with many different types of prompts. The simplest solution to this issue is to break the large templates into smaller ones. For example having an algorithms explanation template and a general explanation template.

L. Lessons Learned

Building and testing the system taught us some practical lessons:

- A few short questions with good defaults are enough to add most of the missing context, and users can still skip them.
- A score is only useful when the tool can also act on it. Every problem the scorer can report should have a matching fix.
- Keyword rules are fast and easy to explain, but they must match whole words, and they need regular testing with new requests.
- A fixed pass mark should be chosen from data. A sweep like the one in Fig. 13 is a simple way to do this.
- Short instructions to the host assistant, such as “use at most two tool calls”, keep the MCP workflow fast and predictable.

M. Threats to Validity

Some things may affect our results. The test requests were written by the project team, so they may be closer to the trigger words than real requests from other users. The intended task type of each request was picked by the team, and someone else might pick differently for mixed requests, such as a business plan for an app. The APEX score comes from the same rules that shape the templates, which favors the improved prompts. The case study used only eight requests, one chatbot, and the team's own judgment. Lastly, all requests were in English. For these reasons, our numbers should be seen as a first check of the design, not as final proof of better answers.

VIII. Applications

The system can be used in a variety of situations:

Education: Students can take a prompt and generate prompts asking for analogies, examples and practice questions of an appropriate difficulty.

Research: Researchers can generate prompts asking for areas of improvement, hypotheses and how to measure outcomes to aid with planning your literature review.

Software Teams: Developers can take a one-liner of the feature and generate prompts asking for architectural decisions, edge cases and tests.

Business Users: Managers can generate prompts asking for comparisons between options, listing any hidden costs and then providing a clear recommendation.

Tool Builders: The detailed approach makes it easy for other programs to use each step of the process (e.g. save the entire history of prompts for the team).

As this tool runs within the assistant you are already using, there is no new learning required.

IX. Ethical Considerations

This should be used carefully. Firstly, giving a stronger prompt will not make the model more truthful. There is still a risk of the model providing inaccurate facts. This means that even with the prompt asking to separate facts and opinions, if there are any key pieces of information (such as health, legal and financial information) they should be checked. Secondly, the model may appear to have more confidence in its answer. Thirdly, the text that you provide will be saved in the session files on your local machine so you should assume that your text is private. Fourthly, the optimizer will insert your text into the prompt. This means that any malicious instructions can be carried through [62]. This tool should be used to help you learn. The students should use the structured answers to help understand the topic.

X. Limitations

- The APEX scorer checks the form of a prompt by counting words. It does not measure the quality of the final answer, and it gives the same points to any audience line.
- The classifier uses ordered keyword lists and matches parts of words. This caused 4 mistakes in 60 requests and 3 mistakes in 5 extra requests.
- The improvement rules do not cover the problems that the scorer reports most often, so the loop stops getting better after one round.
- No prompt ever reached the pass mark of 85. The built-in check in the code also fails, because its first request scores 72.2, below its own limit of 80.
- The granular path stops at stage 9 because of an argument bug in the scoring tool.
- The test set has only 60 English requests, and the case study has only eight requests judged by the project team.
- User text is copied into the prompt without any filtering, which could carry injected instructions [61], [62].

XI. Future Work

We plan several improvements. The classifier will match whole words and, later, use sentence embeddings [33] or a small trained model, with the keyword rules kept as a fast first step. The refiner will get a rule for every problem type, and an optional mode will let a model rewrite the prompt from the scored history, as in OPRO [10]. The scorer will be tuned on a larger set of prompts, and its results will be compared with ratings by people and by a model judge [36]. We also plan to test real answer quality on public benchmarks such as GSM8K [39], Big-Bench Hard [40], and IFEval [65]. Support for Indian languages such as Hindi and Marathi is another goal.

Lastly, we will add input filtering and follow the advice from MCP security studies [58], [59] and prompt injection research [63], [64].

XII. Conclusion

This paper details the development of an Adaptive Universal Prompt Optimizer. This simple tool produces an optimized system prompt, structured user prompt and explanation of how good the output is. It combines concepts from APET, PICCO, MetaSPO, APEX and OPRO. This is done using a single 11-step workflow. It can be used via the terminal or as an MCP server with 17 tools. 60 prompts were tested with the optimizer. The accuracy of the classifier was 93.3%, the average APEX score increased almost double (from 36.4 to 70.9) and optimization was completed in under 0.1ms. A case study demonstrated that these improved prompts produced answers which had clearer explanations of their plans and covered limitations better. However, there were clear issues with the optimizations. The pass criteria is too strict, the refiner cannot improve the most common issues and the classifier is prone to being tricked by the parts of words. Overall we believe that the simplicity of the prompt improvements and the fact that they are self-explanatory will aid students and other non-experts to gain more useful answers. The weaknesses identified provide a clear roadmap for future versions.

References

[1] A. Vaswani et al., "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017.

[2] T. B. Brown et al., "Language models are few-shot learners," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 1877–1901, 2020.

[3] L. Ouyang et al., "Training language models to follow instructions with human feedback," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, 2022.

[4] J. D. Zamfirescu-Pereira, R. Y. Wong, B. Hartmann, and Q. Yang, "Why Johnny can't prompt: How non-AI experts try (and fail) to design LLM prompts," in *Proc. CHI Conf. Human Factors in Computing Systems*, 2023.

[5] M. Sclar, Y. Choi, Y. Tsvetkov, and A. Suhr, "Quantifying language models' sensitivity to spurious features in prompt design or: How I learned to start worrying about prompt formatting," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2024.

[6] M. Mizrahi, G. Kaplan, D. Malkin, R. Dror, D. Shahaf, and G. Stanovsky, "State of what art? A call for multi-prompt LLM evaluation," *Trans. Association for Computational Linguistics*, vol. 12, 2024.

[7] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[8] S. Schulhoff et al., "The Prompt Report: A systematic survey of prompting techniques," arXiv:2406.06608, 2024.

[9] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, "A systematic survey of prompt engineering in large language models: Techniques and applications," arXiv:2402.07927, 2024.

[10] C. Yang, X. Wang, Y. Lu, H. Liu, Q. V. Le, D. Zhou, and X. Chen, "Large language models as optimizers," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2024.

[11] Y. Zhou et al., "Large language models are human-level prompt engineers," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2023.

[12] R. Pryzant, D. Iter, J. Li, Y. T. Lee, C. Zhu, and M. Zeng, "Automatic prompt optimization with 'gradient descent' and beam search," in *Proc. Conf. Empirical Methods in Natural Language Processing (EMNLP)*, 2023.

[13] F. Wang, S. Si, C.-J. Hsieh, and I. S. Dhillon, "APEX: Automated prompt engineering expert with dynamic data selection," arXiv:2606.11459, 2026.

[14] Y. Choi, J. Baek, and S. J. Hwang, "System prompt optimization with meta-learning," arXiv:2505.09666, 2025.

[15] O. Khattab et al., "DSPy: Compiling declarative language model calls into self-improving pipelines," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2024.

[16] Anthropic, "Model Context Protocol specification," 2025. [Online]. Available: https://modelcontextprotocol.io/specification

[17] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, 2022.

[18] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, 2022.

[19] X. Wang et al., "Self-consistency improves chain of thought reasoning in language models," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2023.

[20] S. Yao et al., "Tree of thoughts: Deliberate problem solving with large language models," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 36, 2023.

[21] A. Kong et al., "Better zero-shot reasoning with role-play prompting," in *Proc. Conf. North American Chapter of the Association for Computational Linguistics (NAACL)*, 2024.

[22] M. Zheng, J. Pei, L. Logeswaran, M. Lee, and D. Jurgens, "Is 'a helpful assistant' the best role for large language models? A systematic evaluation of social roles in system prompts," in *Findings of the Association for Computational Linguistics: EMNLP 2024*, 2024.

[23] L. Reynolds and K. McDonell, "Prompt programming for large language models: Beyond the few-shot paradigm," in *Extended Abstracts of the CHI Conf. Human Factors in Computing Systems*, 2021.

[24] S. Mishra, D. Khashabi, C. Baral, Y. Choi, and H. Hajishirzi, "Reframing instructional prompts to GPTk's language," in *Findings of the Association for Computational Linguistics: ACL 2022*, 2022.

[25] J. White et al., "A prompt pattern catalog to enhance prompt engineering with ChatGPT," arXiv:2302.11382, 2023.

[26] S. M. Bsharat, A. Myrzakhan, and Z. Shen, "Principled instructions are all you need for questioning LLaMA-1/2, GPT-3.5/4," arXiv:2312.16171, 2023.

[27] B. Lester, R. Al-Rfou, and N. Constant, "The power of scale for parameter-efficient prompt tuning," in *Proc. Conf. Empirical Methods in Natural Language Processing (EMNLP)*, 2021.

[28] X. L. Li and P. Liang, "Prefix-tuning: Optimizing continuous prompts for generation," in *Proc. Annu. Meeting of the Association for Computational Linguistics (ACL)*, 2021.

[29] A. Prasad, P. Hase, X. Zhou, and M. Bansal, "GrIPS: Gradient-free, edit-based instruction search for prompting large language models," in *Proc. Conf. European Chapter of the Association for Computational Linguistics (EACL)*, 2023.

[30] M. Deng et al., "RLPrompt: Optimizing discrete text prompts with reinforcement learning," in *Proc. Conf. Empirical Methods in Natural Language Processing (EMNLP)*, 2022.

[31] F. Sebastiani, "Machine learning in automated text categorization," *ACM Computing Surveys*, vol. 34, no. 1, pp. 1–47, 2002.

[32] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North American Chapter of the Association for Computational Linguistics (NAACL-HLT)*, 2019.

[33] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using Siamese BERT-networks," in *Proc. Conf. Empirical Methods in Natural Language Processing (EMNLP-IJCNLP)*, 2019.

[34] L. W. Anderson and D. R. Krathwohl, Eds., *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. New York, NY, USA: Longman, 2001.

[35] J. Sweller, "Cognitive load during problem solving: Effects on learning," *Cognitive Science*, vol. 12, no. 2, pp. 257–285, 1988.

[36] L. Zheng et al., "Judging LLM-as-a-judge with MT-Bench and Chatbot Arena," in *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2023.

[37] JSON-RPC Working Group, "JSON-RPC 2.0 specification," 2010. [Online]. Available: https://www.jsonrpc.org/specification

[38] T. Shin, Y. Razeghi, R. L. Logan IV, E. Wallace, and S. Singh, "AutoPrompt: Eliciting knowledge from language models with

automatically generated prompts," in *Proc. Conf. Empirical Methods in Natural Language Processing (EMNLP)*, 2020.

[39] K. Cobbe et al., "Training verifiers to solve math word problems," arXiv:2110.14168, 2021.

[40] M. Suzgun et al., "Challenging BIG-Bench tasks and whether chain-of-thought can solve them," in *Findings of the Association for Computational Linguistics: ACL 2023*, 2023.

[41] Q. Guo et al., "Connecting large language models with evolutionary algorithms yields powerful prompt optimizers," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2024.

[42] C. Fernando, D. Banarse, H. Michalewski, S. Osindero, and T. Rocktäschel, "Promptbreeder: Self-referential self-improvement via prompt evolution," in *Proc. Int. Conf. Machine Learning (ICML)*, 2024.

[43] D. Kepel and K. Valogianni, "Autonomous prompt engineering in large language models," arXiv:2407.11000, 2024.

[44] F. Chowdhury, N. Mihindukulasooriya, N. S. D'Souza, H. Samulowitz, N. Gupta, T. Hanusiak, and M. Kapitonow, "No-cue prompt engineering with zero tuning," in *Proc. IEEE Int. Conf. Data Mining (ICDM), Demo Track*, 2025, arXiv:2601.03130.

[45] K. Opsahl-Ong et al., "Optimizing instructions and demonstrations for multi-stage language model programs," in *Proc. Conf. Empirical Methods in Natural Language Processing (EMNLP)*, 2024.

[46] M. Yuksekgonul et al., "TextGrad: Automatic 'differentiation' via text," arXiv:2406.07496, 2024.

[47] "Prompt generation technical report," arXiv:2607.11326, Jul. 2026.

[48] A. Madaan et al., "Self-Refine: Iterative refinement with self-feedback," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[49] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[50] D. A. Cook, "The PICCO framework for large language model prompting: A taxonomy and reference architecture for prompt structure," arXiv:2604.14197, 2026.

[51] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2023.

[52] T. Schick et al., "Toolformer: Language models can teach themselves to use tools," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[53] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez, "Gorilla: Large language model connected with massive APIs," arXiv:2305.15334, 2023.

[54] Y. Qin et al., "ToolLLM: Facilitating large language models to master 16000+ real-world APIs," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2024.

[55] Q. Wu et al., "AutoGen: Enabling next-gen LLM applications via multi-agent conversation," arXiv:2308.08155, 2023.

[56] G. Li, H. A. A. K. Hammoud, H. Itani, D. Khizbullin, and B. Ghanem, "CAMEL: Communicative agents for 'mind' exploration of large language model society," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[57] X. Luo et al., "Agent Lightning: Train any AI agents with reinforcement learning," arXiv:2508.03680, 2025.

[58] X. Hou, Y. Zhao, S. Wang, and H. Wang, "Model Context Protocol (MCP): Landscape, security threats, and future research directions," arXiv:2503.23278, 2025.

[59] M. M. Hasan, H. Li, E. Fallahzadeh, G. K. Rajbahadur, B. Adams, and A. E. Hassan, "Model Context Protocol (MCP) at first glance: Studying the security and maintainability of MCP servers," arXiv:2506.13538, 2025.

[60] Y. Yang et al., "A survey of AI agent protocols," arXiv:2504.16736, 2025.

[61] F. Perez and I. Ribeiro, "Ignore previous prompt: Attack techniques for language models," in *NeurIPS ML Safety Workshop*, 2022.

[62] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, "Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection," in *Proc. ACM Workshop on Artificial Intelligence and Security (AISec)*, 2023.

[63] E. Debenedetti, J. Zhang, M. Balunović, L. Beurer-Kellner, M. Fischer, and F. Tramèr, "AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents," in *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2024.

[64] X. Chen, J. Zhang, and F. Tramèr, "Learning to inject: Prompt injection via reinforcement learning," arXiv:2602.05746, 2026.

[65] J. Zhou et al., "Instruction-following evaluation for large language models," arXiv:2311.07911, 2023.

[66] W. Cui, J. Zhang, Z. Li, H. Sun, D. Lopez, K. Das, B. A. Malin, and S. Kumar, "A survey of heuristic-based search algorithms for automatic instruction-oriented prompt optimisation," in *Findings of the Association for Computational Linguistics: ACL 2025*, Vienna, Austria, 2025, pp. 22093–22111.