IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511154

Smart Prompts, Smarter Language: AI in Software Engineering

Prapthi H P[1], Ramya B P[2], Ranjan Prajwal[3], Sanjana B[4], K.B. Janaki[5]

Dept. of Computer Science and Engineering, East West College of Engineering, Bengaluru, India[1]

Dept. of Computer Science and Engineering, East West College of Engineering, Bengaluru, India[2]

Dept. of Computer Science and Engineering, East West College of Engineering, Bengaluru, India[3]

Dept. of Computer Science and Engineering, East West College of Engineering, Bengaluru, India[4]

Dept. of Computer Science and Engineering, East West College of Engineering, Bengaluru, India[5]

Abstract: Large Language Models (LLMs) can write code, debug programs and draft documentation, but the quality of their output depends heavily on the prompt, and most non-technical users cannot write specific, constrained and context-rich prompts. This paper presents SmartPrompt (Smart Prompts, Smarter Language), an automated prompt-engineering web platform that sits between the user and the LLM. A rough idea, entered as text, speech or an image, passes through a four-stage meta-prompting pipeline (Discovery, Compose, Critique and Evaluate) that returns three optimized prompt variants, each guided by a different technique from a taxonomy of 18 prompt-engineering techniques and scored on a six-axis rubric. The chosen prompt can be run concurrently on several LLM providers to compare answers and latency side by side. The system learns from use through a scikit-learn intent classifier (HashingVectorizer with SGDClassifier, updated by partial_fit), a Beta-posterior technique selector, rubric-weight regression on user ratings and dynamic few-shot retrieval of user-edited prompts. It is built with React, Vite and Tailwind CSS on the front end and FastAPI with SQLite on the back end, and streams stage progress using Server-Sent Events. In testing, the local learning components added well under 5 ms per request, intent accuracy on 45 held-out requests rose from 46.7% to about 60% with 18–36 labelled examples, and all 22 API cases and 15 automated tests passed.

Keywords: Prompt Engineering, Large Language Models, Meta-Prompting, Intent Classification, Online Learning, Multimodal Input, FastAPI, Software Engineering

I. INTRODUCTION

The rapid advancement of Large Language Models (LLMs) such as GPT-4, Gemini, Claude and open-source alternatives has changed how software is written and maintained. These models can translate natural-language descriptions into working code, localize bugs, explain unfamiliar programs and generate documentation [1], [2]. A persistent obstacle limits their practical value: they depend heavily on the prompt. Small changes in phrasing, ordering or the presence of examples can lead to substantially different outputs [3], [4]. Prompt engineering, the practice of designing and refining input instructions, has therefore become a discipline of its own, and because it manipulates the input rather than the model's parameters it is cheaper and more scalable than fine-tuning [5].

Writing a good prompt is a skill. An expert prompt states a role, describes the audience, lists constraints, specifies the output format and often supplies examples, whereas a novice types something like "make a login page". Vague prompts invite the model to guess missing requirements, producing wrong, inconsistent or hallucinated answers; users then spend time and API credit on trial and error; and domain experts and managers, who know what is needed, are the least equipped to express it. The literature offers many remedies, including prompt catalogs [6], test-driven prompting [2], template-based frameworks [7], evolutionary optimization [8], interpretable prompt metrics [9] and example-selection strategies [10]. However, each is typically evaluated on a single task or dataset, and none is packaged as a tool that an ordinary user can open in a browser.

This paper presents SmartPrompt (Smart Prompts, Smarter Language), an automated prompt-engineering web application that accepts a rough idea as text, speech or an image and returns three optimized, scored and explained prompts. Its main contributions are: (i) a four-stage meta-prompting pipeline (Discovery, Compose, Critique, Evaluate) implemented as immutable server-side prompts and streamed to the browser; (ii) a taxonomy of 18 prompting techniques used as strategy guidance, one technique per variant; (iii) a lightweight online-learning layer made up of a hashed-feature intent classifier, a Beta-posterior technique selector, a rubric-weight regression and dynamic few-shot retrieval; and (iv) a transparent

 This work is licensed under a Creative Commons Attribution 4.0 International License



International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026
DOI: 10.17148/IJARCCE.2026.1511154

interface that exposes intent confidence, technique tags, rationales and six-axis scores, and runs the chosen prompt concurrently on several LLM providers. Section II reviews related work, Section III describes the architecture, Section IV the methodology, Section V the implementation, Section VI the results, and Section VII concludes.

II. LITERATURE REVIEW

A. Foundations of Prompt Engineering

An LLM generates a continuation of its input, and sufficiently large models can perform new tasks from instructions and a few examples given in the prompt without any gradient update, a capability called in-context learning [11]. Zero-shot prompting supplies only an instruction, while few-shot prompting adds input–output examples; Agade et al. note that few-shot and chain-of-thought prompting improve accuracy on complex tasks at the cost of higher token use and sensitivity to example choice [12]. Chain-of-thought (CoT) prompting elicits intermediate reasoning steps [13], and self-consistency samples several reasoning paths and keeps the most consistent answer [14]. Pai catalogs sixteen techniques as reusable prompt patterns [6], an idea reflected in the 18-technique taxonomy used here. Prompting is one of three common ways to adapt an LLM, alongside fine-tuning and retrieval-augmented generation (RAG) [15]. In a chatbot study, Chaubey et al. reported accuracies of 87.8% for fine-tuning, 84.5% for RAG and 83.2% for prompt engineering, noting that prompting is the cheapest to apply [5].

B. Related Studies

Table I summarizes the studies that most influenced the design. Subahi proposes a taxonomy of 18 techniques in four prompt patterns (Discovery, Blueprinting, Refinement and Evaluation) for domain-specific-language development, which is the conceptual ancestor of our pipeline [1]. Rizqullah and Albassam show that test-driven prompting improves Python code-generation accuracy by 6.09% on average across eight LLMs [2]. PEEM defines a nine-axis rubric for jointly evaluating prompts and responses whose scores correlate strongly with accuracy [9]; our Evaluate stage adapts its prompt criteria into a six-axis rubric. Other work covers template-integrated prompting for code analysis [7], few-shot example selection for translation [10], genetic-algorithm prompt evolution that outperformed manual prompts by 24% [8], text-to-image prompt enhancement [16], graphical-abstract prompting [17], best-practice prompt methodology [18] and comparisons of technique combinations across open models [4]. Son and Lee found that the best combination depends on both task and model, which motivates learning technique preferences instead of hard-coding them.

TABLE I COMPARISON OF KEY REVIEWED STUDIES

Study	Method and key result	Limitation
Subahi [1]	18 techniques in four patterns (Discovery, Blueprinting, Refinement, Evaluation)	Manual prompt crafting; monitoring DSLs only
Rizqullah and Albassam [2]	Test-driven prompting; +6.09% average accuracy over eight LLMs	Python and competitive programming only
Hong et al. [9]	PEEM nine-axis rubric with rationales; Spearman $\rho \approx 0.97$ with accuracy	English benchmarks; self-preference bias
Chen and Chen [7]	Templates with CoT for code analysis; F1 = 0.94	C/Pthreads; narrow baselines
Laki and Yang [10]	Few-shot machine translation; 10–25% BLEU gain	One language pair; 100 test segments
Loss and Dhuvad [8]	Genetic-algorithm prompt evolution; +24% over manual prompts	High compute cost; fixed population
Chaubey et al. [5]	Fine-tuning 87.8%, RAG 84.5%, prompting 83.2%	Single dataset; chatbots only
Son and Lee [4]	Technique combinations on three open LLMs	Two benchmarks only

C. Research Gaps

Read together, these studies reveal six recurring gaps: domain specificity, limited evaluation, computational cost, lack of interpretability, fragmentation of techniques, and accessibility barriers for non-experts. Table II maps each gap to the corresponding design response in SmartPrompt.

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511154

TABLE II RESEARCH GAPS AND SMARTPROMPT RESPONSES

Gap	Response in SmartPrompt
Domain specificity	Domain-agnostic meta-prompting; nine intent categories covering code, writing, data analysis, image generation and research
Evaluation limits	Six-axis rubric on every variant; stored scores and feedback allow ongoing evaluation; multi-provider comparison
Computational cost	No evolutionary search; three techniques chosen by a cheap Beta-posterior score, with one four-stage pass per request
Interpretability	Technique tags on each card, rubric details, critique rationales and an intent-confidence display
Fragmentation	Role, few-shot, test-driven, constraint-first and other strategies integrated in one pipeline
Accessibility	Plain-language input by text, voice or image; the user never has to name a technique

III. SYSTEM ARCHITECTURE

SmartPrompt is a three-tier, decoupled web application (Fig. 1). The presentation tier is a React single-page application built with Vite and styled with Tailwind CSS; it consumes streaming responses through the browser's ReadableStream API, parsing each data: line as a JSON event so that stages and results appear progressively. The application tier is a FastAPI service that validates requests with Pydantic, orchestrates the pipeline, calls the learning module, talks to provider SDKs or REST APIs and returns a text/event-stream response. The data tier is SQLite accessed through SQLAlchemy and holds two tables, generations and feedback, so no database server is required. The design follows five principles: decoupling of client and server, streaming-first progress reporting, prompts treated as data (the four meta-prompts and 18 techniques are text constants), cheap local intelligence based on linear models over hashed features, and transparency toward the user.

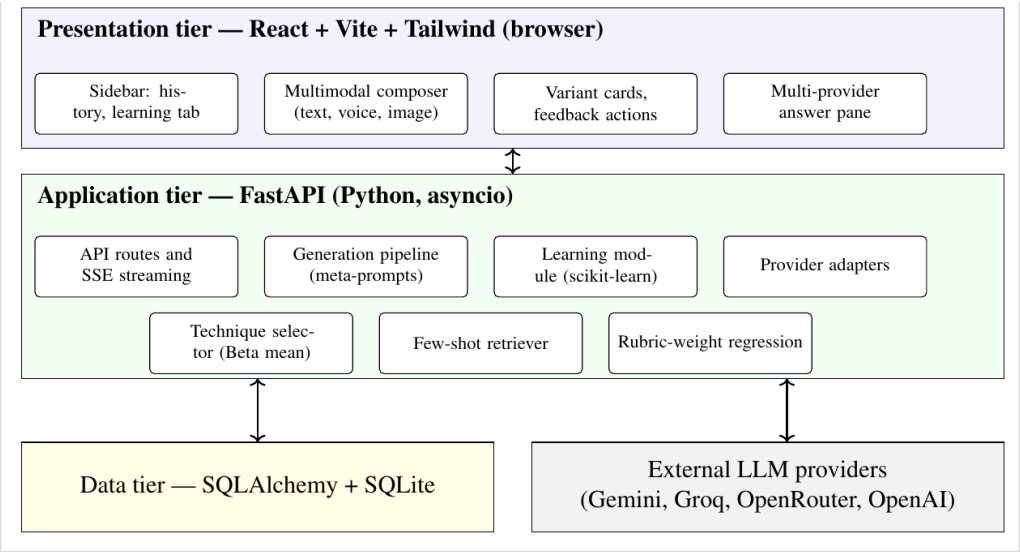


Fig. 1 Layered architecture of SmartPrompt.

IV. METHODOLOGY

A. Four-Stage Meta-Prompting Pipeline

Meta-prompting uses a prompt whose job is to produce or improve other prompts. SmartPrompt defines four meta-prompts as immutable server-side string constants, so clients cannot alter them and each stage has a single, auditable responsibility (Fig. 2). In Stage 1, Discovery, the LLM acts as an analyst and turns the raw text, optional context and any image description into a structured understanding of the hidden goal, target audience and missing constraints, reflecting the Discovery pattern of [1]. In Stage 2, Compose, the LLM writes exactly three prompt variants, each guided by one of three selected techniques, with few-shot examples retrieved from the database injected into every variant. The technique text is guidance, and the system does not verify that a variant follows it exactly. In Stage 3, Critique, the LLM re-reads its drafts for vague wording, unstated assumptions, missing constraints and safety issues, then rewrites each prompt and

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511154

records a rationale; if a critique call fails, the original draft is kept so that the user still receives a result. This stage is an instance of self-refinement [19].

In Stage 4, Evaluate, each refined prompt is scored from 1 to 10 on six axes: clarity and structure, linguistic quality, specificity, constraint coverage, actionability, and fairness and safety. The first, second and sixth axes correspond to the prompt criteria of PEEM [9]. The overall score of variant v is where r(v,k) ∈ [1, 10] is the score of variant v on axis k and w(k) ≥ 0 is the weight of that axis. The weights start equal and are later adjusted from user ratings; the variant with the highest S(v) receives the "Recommended" badge.

$$S_v = \frac{\sum_{k=1}^{6} w_k r_{v,k}}{\sum_{k=1}^{6} w_k} \tag{1}$$

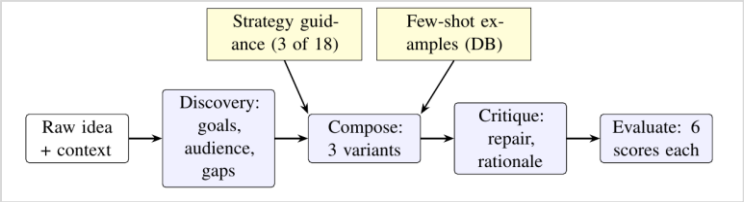


Fig. 2 The four-stage meta-prompting pipeline with its injected inputs.

B. Intent Classification

Before any LLM call, the request is classified locally into one of nine intents: code generation, debugging, requirement clarification, documentation, writing, data analysis, image generation, research and general. Text is vectorized with scikit-learn's HashingVectorizer (2048 features, unigrams and bigrams), which maps each n-gram to a column by a hash function so that no vocabulary is stored and unseen words are handled immediately [20]. An SGDClassifier with logistic loss then gives the class probability normalized across classes, where φ(x) is the hashed feature vector; the reported confidence is the maximum class probability. Because SGDClassifier supports partial_fit, the model is updated one example at a time without retraining from scratch [21]. At start-up it is primed with two seed phrases per intent so that predictions are available from the first request.

$$P(c|x) = \frac{1}{1+e^{-(w_c{}^{\top}\varphi(x)+b_c)}} \tag{2}$$

C. Technique Selection

For each (intent, technique) pair the system stores s successes in n attempts. With a uniform Beta(1, 1) prior, the posterior mean of the success probability is which equals 0.5 with no data, 0.8 for 3 successes in 3 attempts and 0.2 for 0 in 3, and approaches s/n as n grows. A deterministic tie-breaker below 0.01, computed from the character codes of the intent and technique names (their sum modulo 997, divided by 99 700), is added so that techniques with equal scores are ordered consistently rather than always by list position. The three techniques with the highest total score are assigned to the three variants. For example, for debugging with (s, n) = (4, 5), (1, 1) and (0, 0), the posterior means are 0.714, 0.667 and 0.5.

$$\hat{\theta} = E[\theta|s,n] = \frac{1+s}{2+n} \tag{3}$$

D. Feedback and Learning Loop

Four user actions generate feedback: a 1–5 star rating, thumbs up or down, copy, and edit. A variant counts as a success when its rating is at least 3 or, for unrated signals, when the action is thumbs-up, copy or use; counts are recomputed from the persisted feedback table, so they survive restarts. On each feedback event the classifier is updated by partial_fit with the original request and its stored intent, and with the edited prompt when the user changed it; at start-up the generations table is replayed through partial_fit, so no pickled model is needed. Rubric weights are learned by an online ridge regression of the star rating on the six rubric scores (scikit-learn Ridge, α = 1.0): negative coefficients are clipped, the rest are normalized to sum to one, and the weights stay uniform (1/6 each) until at least six rated variants exist. Finally, edited and highly rated prompts are retrieved as dynamic few-shot examples for later requests of the same intent, in the spirit of example-selection work [10].

E. Multimodal Input and Multi-Provider Execution

Voice input uses the browser's Web Speech API, so no audio is sent to the server. An uploaded image (PNG, JPEG or WebP, at most 8 MB) is described by a vision-capable model, and the description enters the pipeline as additional context;

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511154

a background-mode toggle directs the model to describe only a matching environment. When the user selects "Run on LLMs", the chosen prompt is dispatched to the configured providers (Gemini, Groq, OpenRouter and OpenAI) as one asynchronous task per provider consumed with asyncio.as_completed. Each answer is therefore shown with its latency as soon as it arrives, total waiting time is that of the slowest provider rather than the sum, and a failing provider never blocks the others.

V. IMPLEMENTATION

The back end is a Python package with sub-packages for the learning module, the generation pipeline and the provider adapters. Configuration is read from environment variables; the default database is a local SQLite file accessed through SQLAlchemy, and CORS middleware admits the front-end origin. Pydantic models validate requests before any work starts: text must be 2–6000 characters, context at most 4000, star ratings lie in [0, 5], and a provider run must name a variant in [0, 2] and at least one provider. Image uploads other than PNG, JPEG or WebP are rejected with HTTP 400, as are empty files, and files above 8 MB receive HTTP 413. A provider failure during image analysis maps to HTTP 502, while during a provider run it becomes a per-provider failure result.

The /api/generate endpoint returns a StreamingResponse. An asyncio.Queue carries stage names from the pipeline coroutine to the response generator, which emits each as a stage event, stores the final result as a generation row and sends a result event; any exception cancels the task and streams an error event instead of breaking the connection silently. Because these are POST requests, the browser's native EventSource cannot be used, so the client reads the response body with a TextDecoder, splits it at blank lines and parses each data: payload. Table III lists the main endpoints.

TABLE III MAIN REST AND SSE ENDPOINTS

Path	Method	Purpose
/api/generate	POST	Text pipeline; SSE stream of stage, result or error events
/api/generate-from-image	POST	Image pipeline (multipart upload, background-mode flag)
/api/feedback	POST	Store a feedback signal, update learning, return statistics
/api/run-llms	POST	Run a variant's prompt on selected providers; SSE stream
/api/history	GET	Latest 100 generations, optionally filtered by a search string
/api/learning-stats	GET	Totals, average rating, trend and top techniques per intent
/api/memory/reset	POST	Delete all feedback and generations and reset the model

VI. RESULTS AND DISCUSSION

A. Test Summary

Testing combined component checks, the project's own pytest suite and live runs of the application. All 15 automated pytest tests passed (with provider calls mocked), all 17 Pydantic boundary cases behaved as documented, all 22 API cases run against the working application matched their expected results, and all UI scenarios passed. The learning-module checks ran the project's learning code unmodified with an in-memory stub in place of the database session.

B. Intent Classifier

A set of 45 short requests written by the authors (five per intent) was classified by the cold-start model and again after adding labelled examples (Table IV). Accuracy rose from 46.7% to 62.2% after 18 examples and was 60.0% after 36. With 45 test items, one request equals 2.2 percentage points, so the small dip is within noise and no steady improvement is claimed. Mean confidence rose monotonically, and rebuilding the model from the stored generations reproduced the 36-example state (60.0%).

TABLE IV ACCURACY ON 45 HELD-OUT REQUESTS

Model state	Correct	Accuracy	Mean confidence
Seeds only (18 phrases)	21/45	46.7%	0.268
+ 18 labelled examples	28/45	62.2%	0.356
+ 36 labelled examples	27/45	60.0%	0.404
Replay of the same 36 as stored generations	27/45	60.0%	—

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511154

On seventeen cold-start inputs, in-vocabulary requests were classified correctly with confidence 0.55–0.88, whereas out-of-vocabulary ones received lower confidence (0.11–0.58) and several were wrong; for example, a use-case-diagram request was labelled debugging. Low confidence is therefore a useful warning signal that a future version could use to ask the user to confirm the intent. Training the sentence "trip to nepal now" five times with the label general moved its prediction from debugging (0.114) to general (0.936), and a reset restored the original prediction exactly.

C. Technique Selector and Rubric Weights

Six scenarios were run through the selector and compared with an independent re-implementation of the Beta-mean-plus-tie-breaker formula; all six top-three lists were identical. They confirm that good ratings promote a technique, bad ratings demote it below untested ones, an unrated copy counts as a success while an edit does not, one feedback row updates every technique named in its label, and the success threshold is a rating of 3 or more (3.0 counted, 2.9 did not). With fewer than six rated variants the rubric weights stayed uniform at 0.1667. In a recovery test with 30 synthetic rated variants whose rating depended almost only on specificity, the fitted weights were 0.934 for specificity, 0.045 for constraint coverage and 0.020 for clarity and structure, with the other three axes at zero. The data are synthetic, so no claim is made about real users.

D. Performance

Each local operation was called 2000 times (Table V). Local intelligence adds well under 5 ms per request, so end-to-end time is dominated by the four LLM stages and the per-variant critique calls, which were not measured here.

TABLE V MEAN TIME PER CALL OF LOCAL OPERATIONS

Operation	Mean per call
classify (vectorize + predict_proba)	0.30 ms
choose (18 techniques, no feedback)	10 µs
train (one partial_fit)	2.2 ms

E. Application Screenshots

Fig. 3 shows the front page: the heading "What are you trying to do?", suggestion cards for first-time users, and a composer that accepts typed text, voice dictation and an image, with options for extra context, provider execution and background mode. The sidebar lists recent prompts, a Learning panel and the theme switch.

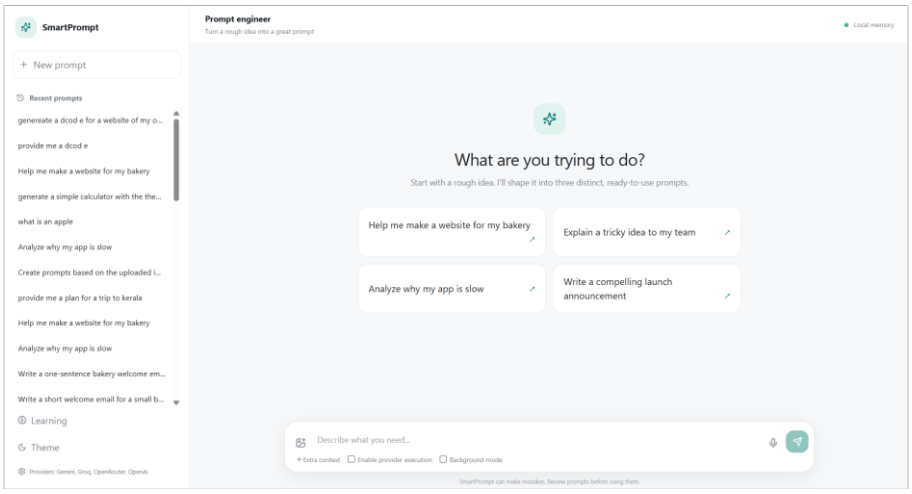


Fig. 3 Front page of the SmartPrompt web application.

Fig. 4 shows the result for the request "generate a code to create a simple calculator with casio theme": three variant cards, each with a technique tag, a rationale, a collapsible score row and feedback controls (copy, edit, thumbs, 1–5 stars). The first card (rubric-driven) is flagged Recommended with a score of 10/10, while the other two (tone control and examples-of-bad-output) score 9/10. In this run the intent label read image generation with 60% confidence although the request is plainly code generation, which illustrates the cold-start weakness discussed above; the prompts remain suitable because the Discovery stage works from the request text itself.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511154

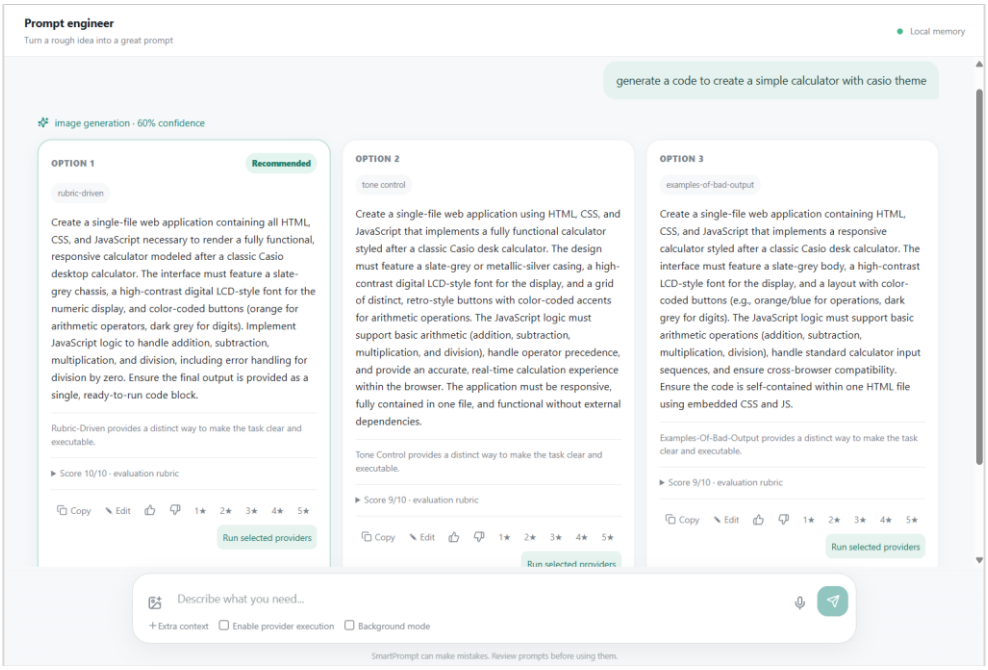


Fig. 4 Three scored prompt variants generated for a calculator request.

Fig. 5 shows the Provider results pane with the Gemini answer to the recommended prompt, listing the provider, the model (gemini-3.1-flash-lite) and a measured latency of 4246 ms. Only one provider is visible in this capture, so it illustrates the per-provider model and latency display rather than a side-by-side comparison, and a single latency value is not a benchmark.

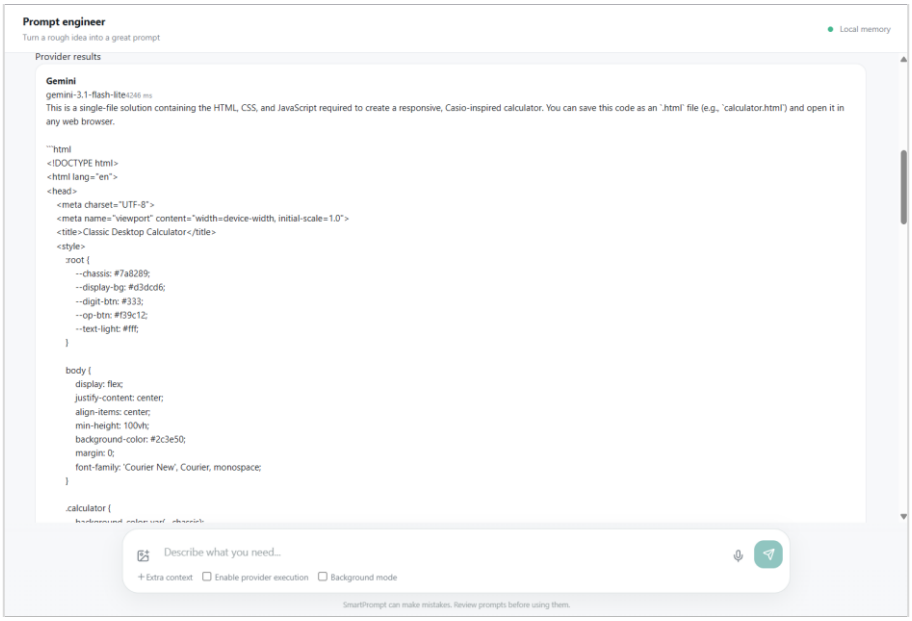


Fig. 5 Provider results pane showing the model name, latency and returned code.

F. Discussion and Limitations

The executed tests support that the local components behave as designed, add negligible latency and match their specification. An intent accuracy of about 60% with 18–36 examples is modest and shows that the classifier depends on accumulated user feedback. Whether the generated prompts lead to better final answers than raw prompts is plausible given the literature (test-driven prompting +6.09% [2], few-shot +10–25% BLEU [10], automated prompts +24% [8]), but those figures belong to other systems and tasks and have not been established for SmartPrompt; a controlled user study is future work. Threats to validity include self-evaluation bias, since the same model family writes and scores the prompts [9], a small test set and personal feedback, label noise from self-training, provider non-determinism and advisory

 This work is licensed under a Creative Commons Attribution 4.0 International License

DOI: 10.17148/IJARCCE.2026.1511154

technique labels. The selector is a deterministic ranking rather than a bandit sampler, so it does not guarantee that all 18 techniques are eventually tried, and an edit does not count as a technique success.

VII. CONCLUSION AND FUTURE WORK

SmartPrompt converts a rough idea, typed, spoken or shown in an image, into three optimized, scored and explained prompts through a four-stage meta-prompting pipeline, and improves with use through lightweight online learning. Its local machine-learning components are fast (about 0.3 ms per classification) and behave as specified, while the improvement in final answers still has to be measured. Planned work includes RAG over user-uploaded documents and code repositories [15], cloud synchronization with authentication, an LLM-as-judge layer to compare provider answers, and cost and token analytics. Improvements suggested by the analysis include Thompson sampling or an upper-confidence bound for exploration, counting edits as successes, technique validators, an independent judge model, user confirmation of low-confidence intents, prompt-injection defences [12], multilingual support [16], and DSL and grammar domain packs that return to the project's origins [1].

ACKNOWLEDGMENT

The authors thank the Principal, Dr. Santhosh Kumar G, and the Head of the Department of Computer Science and Engineering, Dr. Lavanya N L, of East West College of Engineering, Bengaluru, for the facilities provided to carry out this work, and the teaching and non-teaching staff for their support.

REFERENCES

[1]. A. F. Subahi, "Prompt engineering for advancing software engineering: Using generative language models in the development of domain-specific languages," *IEEE Access*, vol. 14, 2026, doi: 10.1109/ACCESS.2026.3663331.

[2]. M. Rizqullah and E. Albassam, "Model-agnostic empirical evaluation of test-driven prompt engineering on improving accuracy and efficiency in large language models Python code generation," *IEEE Access*, vol. 14, 2026, doi: 10.1109/ACCESS.2026.3662817.

[3]. G. Beri, "Advanced techniques in prompt engineering for large language models: A comprehensive study," in *Proc. IEEE 4th Int. Conf. ICT Bus. Ind. Gov. (ICTBIG)*, 2024, doi: 10.1109/ICTBIG64922.2024.10911672.

[4]. M. Son and S. Lee, "Performance analysis of prompt-engineering techniques for large language model," in *Proc. IEEE Int. Conf. Consum. Electron. (ICCE)*, 2025, doi: 10.1109/ICCE63647.2025.10930066.

[5]. H. K. Chaubey, G. Tripathi, R. Ranjan, and S. K. Gopalaiyengar, "Comparative analysis of RAG, fine-tuning, and prompt engineering in chatbot development," in *Proc. Int. Conf. Future Technol. Smart Soc. (ICFTSS)*, 2024, doi: 10.1109/ICFTSS61109.2024.10691338.

[6]. A. G. Pai, "Enhancing interaction with large language models: A catalog of prompt engineering techniques," in *Proc. Int. Conf. Comput. Technol. (ICOCT)*, 2025, doi: 10.1109/ICOCT64433.2025.11118417.

[7]. W.-C. Chen and S.-Y. Chen, "TIPS: A prompt engineering framework for code classification and generation on resource-constrained systems," *IEEE Access*, vol. 14, 2026, doi: 10.1109/ACCESS.2026.3658502.

[8]. L. A. Loss and P. Dhuvad, "From manual to automated prompt engineering: Evolving LLM prompts with genetic algorithms," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, 2025, doi: 10.1109/CEC65147.2025.11043059.

[9]. H. K. Hong, E. Lee, S. Park, and J. Kim, "PEEM: Prompt engineering evaluation metrics for interpretable joint evaluation of prompts and responses in LLMs," *IEEE Access*, vol. 14, 2026, doi: 10.1109/ACCESS.2026.3679809.

[10]. L. J. Laki and Z. G. Yang, "Improving machine translation capabilities by fine-tuning large language models and prompt engineering with domain-specific data," in *Proc. IEEE 3rd Conf. Inf. Technol. Data Sci. (CITDS)*, 2024, doi: 10.1109/CITDS62610.2024.10791375.

[11]. T. B. Brown et al., "Language models are few-shot learners," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 33, 2020, pp. 1877–1901.

[12]. N. Agade, S. Hajare, D. Timande, J. Abhyankar, N. Pardhi, and N. U. Sambhe, "Prompt engineering for large language models: Zero-shot, few-shot, and beyond," in *Proc. 3rd DMIHER Int. Conf. Artif. Intell. Healthcare, Educ. Ind. (IDICAIHEI)*, 2025, doi: 10.1109/IDICAIHEI65991.2025.11378986.

[13]. J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 35, 2022, pp. 24824–24837.

[14]. X. Wang et al., "Self-consistency improves chain of thought reasoning in language models," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2023, arXiv:2203.11171.

[15]. P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 33, 2020, pp. 9459–9474.

 This work is licensed under a Creative Commons Attribution 4.0 International License



International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026
DOI: 10.17148/IJARCCE.2026.1511154

[16]. N. V. S. Teja, K. Kumar, and M. Malarvel, "Multilingual text enhancer (MLTE): A LLaMA2 based model for prompt generation," in *Proc. 3rd Int. Conf. Appl. Artif. Intell. Comput. (ICAAIC)*, 2024, doi: 10.1109/ICAAIC60222.2024.10575122.
[17]. M. Ohashi, M. Oka, and K. Ozawa, "An initial study on prompt engineering for automated generation of graphical abstracts," in *Proc. 18th Int. Congr. Image Signal Process., BioMed. Eng. Informat. (CISP-BMEI)*, 2025.
[18]. R. Deshmukh, S. Gurav, R. Raut, Y. Patil, and M. Bhavsar, "Optimizing human-AI interaction: Innovations in prompt engineering," in *Proc. 3rd Int. Conf. Intell. Data Commun. Technol. Internet of Things (IDCIoT)*, 2025, doi: 10.1109/IDCIOT64235.2025.10914815.
[19]. A. Madaan et al., "Self-Refine: Iterative refinement with self-feedback," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 36, 2023, pp. 46534–46594.
[20]. K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature hashing for large scale multitask learning," in *Proc. 26th Int. Conf. Mach. Learn. (ICML)*, 2009, pp. 1113–1120.
[21]. F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

 This work is licensed under a Creative Commons Attribution 4.0 International License