

Making Prompts First-Class Citizens for Adaptive LLM Pipelines

Uğur Çetintemel
Brown University
ugur_cetintemel@brown.edu

Shu Chen
Brown University
shu_chen@brown.edu

Alexander W. Lee
Brown University
alexander_w_lee@brown.edu

Deepti Raghavan
Brown University
deeptir@brown.edu

Duo Lu
Brown University
duo_lu@brown.edu

Andrew Crotty
Northwestern University
andrew.crotty@northwestern.edu

ABSTRACT

Modern LLM pipelines increasingly resemble complex data-centric applications: they retrieve data, correct errors, call external tools, and coordinate interactions between agents. Yet, the central element controlling this entire process—the prompt—remains a brittle, opaque string that is entirely disconnected from the surrounding program logic. This disconnect fundamentally limits opportunities for reuse, optimization, and runtime adaptivity.

In this paper, we describe our vision and an initial design of SPEAR (*Structured Prompt Execution and Adaptive Refinement*), a new approach to prompt management that treats prompts as first-class citizens in the execution model. Specifically, SPEAR enables: (1) structured prompt management, with prompts organized into versioned views to support introspection and reasoning about provenance; (2) adaptive prompt refinement, whereby prompts can evolve dynamically during execution based on runtime feedback; and (3) policy-driven control, a mechanism for the specification of automatic prompt refinement logic as *when-then* rules.

By tackling the problem of runtime prompt refinement, SPEAR plays a complementary role in the vast ecosystem of existing prompt optimization frameworks and semantic query processing engines. We describe a number of related optimization opportunities unlocked by the SPEAR model, and our preliminary results demonstrate the strong potential of this approach.

1 INTRODUCTION

As LLMs are increasingly embedded in real-world systems, prompts have become the primary means of encoding user intent. They steer data retrieval, direct output generation, recover from runtime errors, and coordinate tool use. However, despite their centrality, prompts are often crafted manually in an unrigorous trial-and-error approach, and most LLM pipelines fail to appropriately version or track prompt evolution in a systematic fashion. Even worse, they typically remain as static, opaque strings, completely detached from the broader program logic.

At the same time, these pipelines are rapidly evolving into complex data-centric applications that involve interactions with knowledge bases, conditional fallback, data validation, and multi-agent orchestration. Several popular frameworks (e.g., LangChain [2], DSPy [6]) allow developers to easily construct LLM processing pipelines, and semantic query engines [8, 10, 12, 13] provide a

declarative interface to LLM-augmented query processing. Yet, in all of these approaches, prompt logic exists completely outside the system, a black box that stays entirely hidden to the optimizer and execution engine during runtime.

This paper proposes a radical shift in abstraction: treat prompts as structured data within the system. We introduce our vision and an early design of SPEAR (*Structured Prompt Execution and Adaptive Refinement*), which makes three core contributions:

- **Structured Prompt Management.** Prompts are treated as rich, structured data objects and organized into named views, which promotes modularity and reusability of program logic. Prompt views are also versioned so that developers can introspect them and track their evolution over time. We define a flexible prompt algebra for creating and refining prompts within an application, which facilitates end-to-end optimization of entire pipelines.
- **Adaptive Prompt Refinement.** In SPEAR, prompts are not static templates tuned offline but rather part of the system state that evolves dynamically at runtime. Refinements are operators that modify prompt logic, potentially based on signals available only during execution. SPEAR allows for a range of refinement modes from fully manual to completely automated, giving developers fine-grained control over the whole refinement process.
- **Policy-Driven Control:** SPEAR also introduces *when-then* policies that govern how and under what circumstances to trigger automatic refinements. These policies can apply refinements either before execution based on operator inputs (e.g., selecting a task-specific prompt view) or adaptively after execution by considering runtime feedback (e.g., result confidence, latency threshold, shared token budget). SPEAR’s policy layer unifies static planning and runtime adaptivity, establishing a principled mechanism for application-level refinement control.

Although DSPy [6] aims to replace manual prompt engineering with automated optimization, it operates primarily offline and offers limited runtime refinement capabilities [3, 15]. As such, SPEAR addresses the complementary dimension of runtime prompt refinement, allowing adaptivity of prompt logic based on real-time signals and feedback. This perspective surfaces completely new opportunities for end-to-end optimization, and our preliminary results demonstrate the promise of this approach.

2 SPEAR

This section provides a high-level overview of our approach to prompt management, called SPEAR (*Structured Prompt Execution and Adaptive Refinement*). We begin with a running example to highlight the pain points of current approaches and then introduce each of the key concepts underlying SPEAR’s design.

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution, provided that you attribute the original work to the authors and CIDR 2026. 16th Annual Conference on Innovative Data Systems Research (CIDR ’26). January 18-21, Chaminade, USA

2.1 Running Example

Consider an application that uses LLMs to help a clinician review a patient’s medical history while forming a diagnosis and developing a corresponding treatment plan. One pipeline might focus on extracting and reasoning about clinical notes mentioning medications like enoxaparin, a commonly prescribed anticoagulant.

To implement this pipeline, a developer would likely begin with a straightforward prompt: *“Summarize the patient’s medication history and highlight any use of enoxaparin.”* After observing that initial outputs are inconsistent, with some omitting dosage and others missing timing or administration information, the developer might then attempt to improve the results manually by adding to the prompt: *“Be specific about dosage and indicate whether enoxaparin was administered within the last 48 hours.”* As the pipeline becomes more complicated, the developer will need to begin maintaining prompt variants to apply in specific scenarios (e.g., additional information about drug interactions for inpatients) while also handling low-confidence responses and other exceptions. Although a variety of existing frameworks (e.g., LangChain [2], DSPy [6]) attempt to alleviate some of this burden, the developer is ultimately left with the manual and labor-intensive task of updating the pipeline via brittle ad hoc prompt edits.

SPEAR adopts a fundamentally different perspective: prompts are rich, structured data objects that evolve over time and can be directly manipulated based on runtime signals during execution. As such, SPEAR blurs the line between the program (i.e., prompt logic) and the data upon which it operates, allowing developers to write pipelines that can directly reason over and optimize their own prompt strategies.

2.2 Structured Prompt Management

As mentioned, SPEAR’s key insight is to elevate prompts to first-class citizens by treating them just like any other data in the system. In SPEAR, prompts are not simple strings but rather structured data objects that, in addition to the actual prompt text, store other metadata like runtime statistics and version history as the prompt evolves over time. In the following, we describe how prompts are defined, used, and finally compiled to an executable prompt algebra.

Prompt Views. Like views in SQL, SPEAR represents each prompt as a logical view that encapsulates a reusable piece of program logic expressed in natural language. For instance, in the example from §2.1, the developer would define an `enoxaparin_summary` view that could then be invoked anywhere in the application requiring that information.

Also similar to SQL views, prompt views are composable, such that they are definable in terms of other named views. Developers can therefore decompose complex prompt logic into small, modular building blocks. For example, suppose that `enoxaparin_summary` required information about drug interactions for inpatients with ongoing treatment; in this case, the developer could refine the prompt to conditionally use the `check_interactions` view based on patient status. We cover prompt refinement in §2.3.

Lastly, SPEAR supports prompt view parameterization, allowing developers to define a prompt template that is invoked with specific values at runtime. Continuing to build on the running example, the developer could replace `enoxaparin_summary` with a generic

`medication_summary` prompt view that takes the medication name as a parameter, since the application will likely need to provide similar summaries for many different medications.

Unlike other approaches, prompt views offer key advantages from a developer perspective in terms of code modularity, reusability, and overall maintainability. This abstraction also presents a range of optimization opportunities, which we discuss in §3.

SPEAR maintains the set of all prompt views P as a key-value mapping of named views to prompt objects. Likewise, the context store C contains all runtime data over which prompt views operate, including inputs, outputs, and intermediate results. Prompt views may additionally reference or update the runtime metadata M , a collection of control signals and diagnostic information generated during execution (e.g., shared token budget, latency threshold). Taken together, the prompt views P , context store C , and runtime metadata M comprise the system state.

Prompt Algebra. At the heart of SPEAR is an executable prompt algebra that operates on the system state in a structured and principled way. SPEAR’s algebra is closed under composition, in that every operator must consume and produce a triple of the form (P, C, M) . This design naturally makes the set of operators both flexible and easily extensible.

SPEAR has several built-in core operators, the most important of which is GENERATE (GEN). Using the specified LLM, GEN applies a prompt view to any referenced data in C and stores the result back into C . Other core operators include RETRIEVE (RET), which fetches data from an external data source (e.g., database, web search, API) into C , and SWITCH, which encodes conditional logic for pipeline control flow. In §2.4, we show how the end-to-end pipeline we have built up as part of our running example compiles to a plan with these three operators—including additional prompt refinement logic, which we discuss next.

2.3 Adaptive Prompt Refinement

So far, we have mainly described how developers build pipelines that update the context store C and runtime metadata M . However, since prompts are treated exactly the same as all other data in SPEAR, operators can also directly introspect, reason about, and modify prompt logic. Importantly, this capability extends to dynamic refinement at runtime that can adapt to real-time execution feedback, which sets SPEAR apart from existing frameworks that focus mostly on offline prompt optimization.

More formally, a refinement is any operator that updates the set of prompt views P by modifying existing prompts or composing entirely new ones, which results in a REFINE (REF) edge in the corresponding prompt algebra plan. These refinements might update an existing prompt by appending examples, injecting hints, or rewriting confusing instructions. Next, we explain the different refinement modes in SPEAR that let the developer balance control and automation. We then introduce a policy-based mechanism that allows the developer to specify how SPEAR should apply refinements automatically.

Refinement Modes. Depending on the degree of oversight that the developer desires, SPEAR supports a range of refinement modes that fall into three broad categories:

```

medication_summary = "Summarize the patient's medication history and highlight any use of {medication}."
medication_summary += " Be specific about dosage and indicate whether {medication} was administered within the last 48 hours."
medication_summary += " " + check_interactions(medication) if is_inpatient else ""
output = medication_summary(clinical_notes, medication='enoxaparin', is_inpatient=True)

```

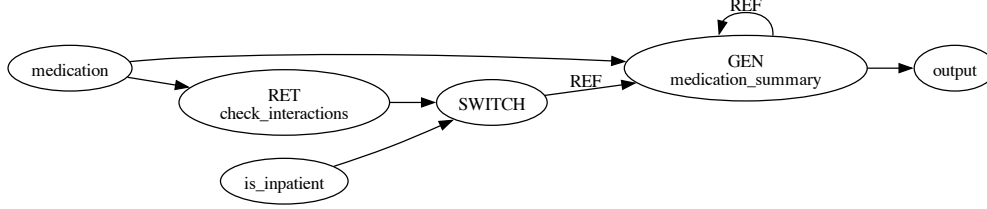


Figure 1: Final implementation of the running example from §2.1, including developer code (top) and compiled plan (bottom).

- **Manual.** This mode is useful when the developer requires low-level control over the refinement (e.g., enforcing a specific output format) or must inject relevant domain knowledge into the prompt. We have already shown several manual refinements applied to the running example from §2.1, including the addition of a request for specific details in the output summary and updating the prompt to accept the medication as a parameter. In all of these cases, the developer explicitly modified the prompt text.
- **Assisted.** This mode offers a more declarative refinement option, where the developer only needs to provide a high-level description of intent. For example, an assisted alternative to the original manual refinement of the enoxaparin prompt shown in §2.1 might look like: “Be sure to include details about recent enoxaparin administration.” In our prompt algebra, this assisted refinement would compile to a GEN operator that consumes the original prompt and produces a new version, which would ideally look similar to the manual refinement.
- **Automatic.** This mode passively monitors the system state at runtime and triggers a refinement based on a user-specified condition. As with assisted refinements, SPEAR will compile the refinement to a prompt algebra plan for execution, all without any developer intervention. We will revisit automatic refinements, including concrete examples and how to specify them, when we introduce policy-driven control.

In practice, these modes often coexist and can even be combined depending on risk tolerance or system maturity. For example, an application might begin with manual refinement during early development, transition to assisted prompting based on observed patterns, and ultimately move toward automatic handling for scale and responsiveness. Conversely, in deployed settings, pipelines may default to automatic refinement, escalate to assisted repair when needed, and fall back to manual oversight in ambiguous or high-risk cases. SPEAR does not prescribe a particular refinement mode but rather provides the flexibility to choose different modes based on application needs and developer preferences.

Policy-Driven Control. As noted, both the manual and assisted refinement modes rely on active developer involvement to determine when to update a prompt. In other words, these types of refinements are applied statically at compile time in response to an explicit command. Automatic refinements, though, are triggered during runtime in response to the current system state.

In SPEAR, refinement policies allow the developer to specify the circumstances in which to apply an automatic refinement to a prompt. Specifically, we define a policy as a simple *when-then*

rule of the form: WHEN condition THEN refinement. As the name suggests, the developer can include one or more refinement actions in refinement. Signals referenced by condition, which actually triggers the refinement, may originate from the context store C , runtime metadata M , or even a prompt’s own metadata (e.g., repeated failed retries, degradation in accuracy). Depending on what system state condition references, the application of refinement can occur either before or after the evaluation of the prompt algebra plan for the corresponding view:

- **Input Condition (Before).** When condition depends only on inputs to an operator (e.g., parameters, materialized intermediate results), SPEAR can apply all refinements before the operator is even evaluated. For instance, the `check_interactions` refinement from the running example can be determined solely based on a patient’s status before any execution takes place.
- **Output Condition (After).** On the other hand, if condition must examine the output of an operator, then SPEAR can only apply refinements after executing that operator. For example, if the output from `medication_summary` does not include all of the necessary information as determined by an LLM judge, the policy could automatically issue a retry or fall back to a different LLM/prompt combination.

While the system state captured in (P, C, M) constitutes SPEAR’s data plane, refinement policies can be thought of as the control plane that determines how and when pipelines automatically evolve. However, they should not be thought of as a fixed set of hard-coded rules. In fact, refinement policies can even be implemented as prompts, meaning that they compile directly into SPEAR’s prompt algebra and can themselves be adaptively refined. For example, policies that trigger infrequently or rarely improve output quality can be deprioritized or pruned, whereas those that consistently provide significant benefits can be promoted. As such, policy-driven refinement control could lay the foundation to build a fully tunable and self-improving control plane with SPEAR.

2.4 Putting It All Together

Returning to the example from §2.1, Fig. 1 shows the complete, end-to-end implementation of the LLM pipeline for generating medication summaries from clinical notes. The developer code (top) begins with a definition of the parameterized `medication_summary`, followed by the manual refinement concerning required content for the output. The next line shows the refinement policy that conditionally applies the `check_interactions` view for inpatients. Until now, the developer has defined several operators but not actually

invoked any of them; note that SPEAR evaluates pipelines lazily, so they exist only as unrendered templates at this point.

On the next line, the developer calls `medication_summary` with the arguments `medication='enoxaparin'`, `is_inpatient=True`, and a specific patient's `clinical_notes` as context. This invocation causes SPEAR to compile `medication_summary` into the executable prompt algebra plan shown in Fig. 1 (bottom), which contains two named inputs, one named output, and three SPEAR operators. In the plan, GEN will call an LLM to produce output for the main prompt view, `medication_summary`. SWITCH implements refinement policy logic that will conditionally apply the previously defined `check_interactions` view, instantiated here as a RET operation from a database of known drug interactions.

The two REF edges in the plan denote each of the refinements applied to the `medication_summary` view. As mentioned, some refinements can be resolved entirely at compile time, such as the REF that appended additional instructions to the original prompt to ensure that the summary would contain specific details. On the other hand, SPEAR must apply the second REF, which depends on an input value, conditionally at runtime.

The developer is now ready to either execute the compiled plan as a standalone pipeline or embed it into an orchestration framework like LangChain for deployment.

3 OPTIMIZATION OPPORTUNITIES

The combination of structured prompt management and adaptive prompt refinement in SPEAR unlocks a range of optimization opportunities. In this section, we highlight examples of such optimizations that a system adopting SPEAR's approach could implement.

Prefix Caching & Reuse. Reusing attention states is a common technique to accelerate prompt evaluation [11]. Since large parts of the prompt may remain unchanged across successive invocations in many pipelines, SPEAR could identify stable portions of a prompt and cache them for reuse in the future. When making only a small refinement to a prompt (e.g., adding an example), SPEAR will usually append the delta to the end of the prompt view rather than rebuilding the entire prompt from scratch. This incremental construction enables straightforward token-level reuse with LLMs that support a KV cache [1] or FlashAttention [4], reducing latency and compute costs. To generalize this idea, SPEAR could employ a structured prompt cache [5] that indexes prompt views and their rendered forms, facilitating efficient reuse across retries or invocations with different arguments.

Input Batching. When the same prompt view is applied repeatedly to many different input items (e.g., invoking `medication_summary` for a variety of different medications), SPEAR could instead execute them as a single batched GEN operation, allowing the LLM to process all inputs in one call. This approach is common in semantic query processing engines [8, 12, 13], where identical query logic is applied concurrently across batches of inputs to maximize throughput by filling as much of the LLM context window as possible. However, input batching also introduces risks like prompt crowding, degraded accuracy due to cross-input interference, and challenges in isolating errors. SPEAR will need to mitigate these challenges by carefully tuning the batch size to balance performance gains with output quality based on workload characteristics, model behavior, and

downstream quality requirements, as well as providing a fallback option to reprocess inputs that cause issues.

Prompt Batching. Similar to input batching, the main idea is to exploit redundancy in the structure of prompts. This situation occurs frequently in SPEAR, since prompt views are constructed from modular and reusable building blocks. Prompt batching submits groups of similar prompts in a single LLM call, amortizing dispatch overhead while enabling parallel inference without significant risk of cross-prompt interference. When prompts share a common prefix, they may additionally benefit from the previously described prefix caching technique, further improving efficiency.

Operator Fusion. SPEAR can support runtime operator fusion via refinements, updating operators to merge them into a single execution unit that can improve efficiency and reduce the size of intermediate result storage. For tightly coupled operators, such as prompt views that operate over different parts of the same input, operator fusion is a natural optimization. However, unlike in traditional settings where operator fusion is almost always beneficial, fusing prompts can actually result in worse accuracy or higher latency. As such, striking the right balance when deciding which or how many operators to fuse in SPEAR presents an interesting opportunity for further exploration.

4 PRELIMINARY RESULTS

Experimental Setup. We conducted all experiments on a server with a single NVIDIA Quadro RTX 8000 GPU (48 GB GDDR6) running a local Ollama instance for model hosting. Our results include an evaluation of four different model sizes from Google's Gemma 3 [17] model family: 1B, 4B, 12B, and 27B. We estimated approximate inference costs for each model from publicly available data, which we report in Table 1.

To evaluate SPEAR, we selected EHRNoteQA [7], a question-answering benchmark based on clinical notes from real-world electronic health records. The benchmark, which contains 962 unique multiple-choice questions with options A–E, mimics the types of pipelines described in the running example from §2.1. For an answer to be marked correct, the prediction must be a single letter (after stripping whitespace) that exactly matches (including capitalization) the ground truth label. Each experiment was repeated three times on the same randomly chosen 200 samples, with mean values reported for accuracy and total cost.

Comparison Points. We compare SPEAR against: (1) *Baseline*, a minimal prompt with only basic instructions for answering JSON-formatted multiple-choice questions; (2) *DSPy*, using the Predict module to manually extract relevant fields from the JSON object; and (3) *DSPy-CoT*, which uses DSPy's built-in chain-of-thought module. For SPEAR, we include the following refinements:

- **R1 (manual, static).** Extends the baseline by adding explicit instructions about output formatting.
- **R2 (assisted, static).** Adjusts the input structure of *R1* by extracting the question, answer choices, and clinical notes from the JSON object into clearly delineated sections with explicit labels, as suggested offline by the 27B model.
- **R3 (assisted, static).** Slightly rearranges the prompt components of *R2* to place the clinical evidence first, followed by the question

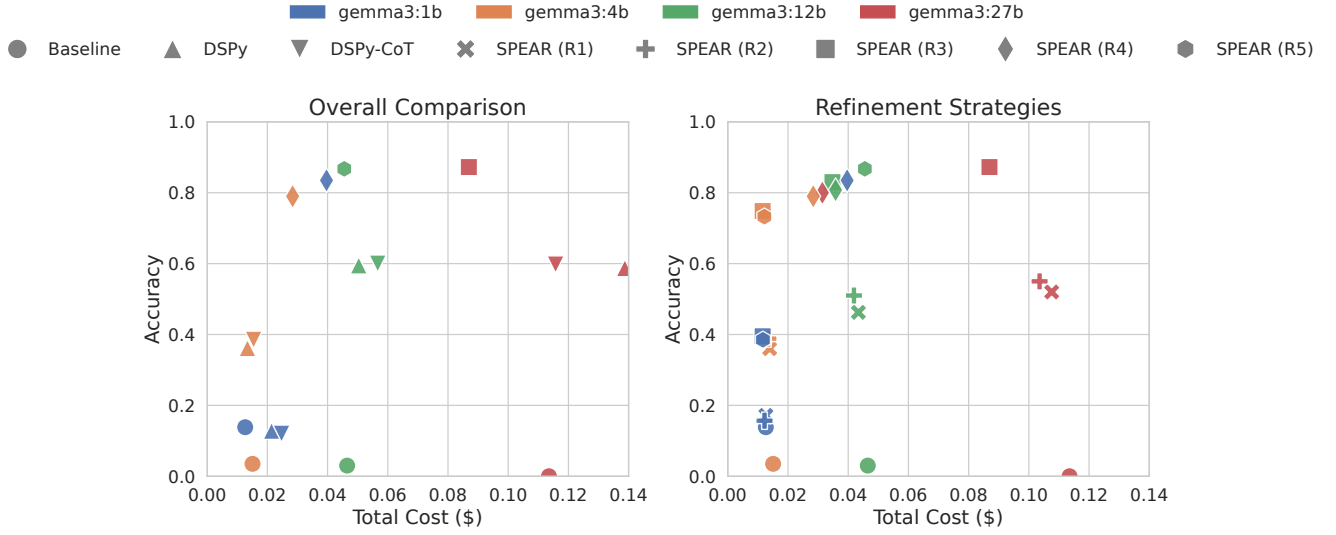


Figure 2: Results of our evaluation, including overall comparison (left) and refinement strategy breakdown (right).

Table 1: Estimated inference costs for each model.

Model	Input (\$/1M)	Output (\$/1M)
gemma3:1B	0.02	0.02
gemma3:4B	0.02	0.04
gemma3:12B	0.06	0.12
gemma3:27B	0.15	0.30

and answer choices, and ending with the output constraints, again as suggested offline by the 27B model.

- **R4 (automatic, input condition).** Based on the routing model’s assessment of the question’s difficulty, routes the question to an appropriate model and then uses R3.
- **R5 (automatic, output condition).** Adds an additional answer option to R3 that allows the model to select *F* when uncertain, then cascades to the next larger model until reaching the largest, which uses R3.

Discussion. Fig. 2 shows the results, with the overall comparison on the left (including the best SPEAR strategy for each model) and a detailed breakdown of SPEAR refinement strategies on the right. We observe several key trends:

- **Comparison to DSPy.** *DSPy* and *DSPy-CoT* provide a substantial improvement over the baseline, but they are consistently outperformed by many of the SPEAR refinements. Despite attempting to enforce the output constraint via a `Literal[...]` field, *DSPy* often struggles to strictly constrain the generation of chatty models in long-context scenarios, leading to parsing errors similar to the baseline. Furthermore, *DSPy*’s internal prompt optimization process is opaque, making it difficult to diagnose and debug specific failure modes compared to SPEAR’s explicit, transparent approach to prompt refinement, which allows for much more targeted fixes.

- **Impact of Prompt Structure.** We observe a clear progression in performance through our refinement stages. The baseline prompt performs poorly mainly because most models fail to output a single letter as an answer without explicit instructions to do so. They often generate verbose reasoning chains that complicate answer extraction and lead to frequent formatting errors, a problem that is even more noticeable with larger models. R3 exhibits the most dramatic leap in performance across every model, which highlights the value of our structured and transparent refinement process; unlike opaque optimization methods, we can explicitly identify and apply interpretable changes like prompt view component reordering.

- **Efficiency of Automatic Refinements (R4, R5).** The automatic refinement strategies successfully optimize the trade-off between cost and accuracy by exploiting the heterogeneity of question difficulty in the benchmark. In particular, R4 demonstrates consistently strong performance by dynamically matching model capacity to query complexity. Notably, using the smallest 1B model as a router yields an impressive accuracy of 83.5%, matching the performance of the standalone 12B model but with the flexibility to call larger models only when needed. This result confirms that a significant portion of the dataset consists of relatively easy questions that do not require the larger, more expensive models to get a correct answer. For R5, a cascade starting with the 12B model achieves 86.7% accuracy, comparable to the standalone 27B model (87.3%) but at nearly half the cost (\$0.046 vs. \$0.087). By only escalating queries when the smaller model explicitly signals uncertainty about its output, this automatic refinement ensures that more expensive model calls are reserved solely for the most challenging questions.

Although techniques like R4 and R5 are not entirely novel, SPEAR enables developers to implement these complex optimization strategies in a simple and modular fashion. By treating them as composable refinement primitives, SPEAR lays the groundwork for reusing

them as building blocks in more sophisticated optimization strategies. Furthermore, this structured approach can allow SPEAR to adaptively refine the routing logic or output uncertainty thresholds based on accumulated runtime feedback.

5 RELATED WORK

Prompt Programming Frameworks. Numerous recent frameworks support the creation of complex LLM pipelines and agentic workflows through chaining, orchestration, and dynamic feedback loops (e.g., LangChain [2]). However, they commonly treat prompts as static strings, with little support for structured prompt management, introspection, or optimization. In contrast, SPEAR promotes prompts to first-class entities with an executable prompt algebra that supports reuse and adaptive runtime refinement.

Manual Prompt Refinement. Prompt engineering techniques, such as adding explicit instructions, incorporating domain-specific examples, or restructuring task descriptions, have been shown to significantly improve output accuracy, consistency, and alignment with task goals across a wide range of applications [16, 18]. For instance, approaches like MedPrompt [9] demonstrate that structured prompts can significantly improve accuracy and relevance in clinical LLM pipelines. SPEAR's refinements aim to support the straightforward application of similar techniques to prompt logic in an easily controllable fashion.

Automated Prompt Optimization. Frameworks like DSPy [6] allow developers to provide high-level specifications in order to automate the prompt optimization process. DSPy Assertions [15] allow for the specification of constraints on model outputs and enable assertion-driven self-refinement both offline and at runtime, whereas techniques like GEPA [3] refine prompts offline by reflecting on execution trajectories to iteratively propose and select improved prompt candidates. SPADE [14] also maintains prompt histories from user edits for the purpose of synthesizing assertions before deployment. Despite some runtime capabilities, these systems mainly target offline prompt optimization, positioning SPEAR as a complementary approach that can help dynamically refine prompts based on feedback during execution.

Semantic Query Engines. Recent LLM-augmented semantic data processing systems [8, 10, 12, 13] offer strong data-level semantics, but their control over LLM behavior is limited; that is, prompt logic is still inlined or statically defined and cannot adapt dynamically during execution. SPEAR fills a different role and could even be paired with these systems to handle runtime prompt refinement.

6 CONCLUSION & FUTURE WORK

This paper described SPEAR, our new approach to prompt management based on the core tenet that prompts should be elevated from simple strings to first-class citizens in LLM pipelines. In doing so, SPEAR unifies structured prompt management, adaptive refinement, and policy-driven control into a coherent execution model. Our approach enables richer forms of prompt introspection, principled reuse, and dynamic optimization, all of which are difficult or impossible in existing frameworks. Early results suggest that this perspective opens the door to a broad class of optimizations that can substantially improve robustness, efficiency, and developer productivity. In future work, we plan to continue to develop SPEAR

and further explore the many fruitful optimization opportunities we highlighted throughout the paper.

ACKNOWLEDGMENTS

This work is supported by a Brown Seed Grant and the National Science Foundation (NSF) Graduate Research Fellowship Program (GRFP) under grants 2439559 and 2040433. Additional support was provided through Google's Research Scholar Program. We also thank the members of the VectraFlow research team for their continued insights and constructive feedback about the project.

REFERENCES

- [1] 2025. Automatic Prefix Caching. https://docs.vllm.ai/en/stable/features/automatic_prefix_caching/.
- [2] 2025. LangChain. <https://www.langchain.com/>.
- [3] Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziemis, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Daniel Klein, Matei Zaharia, and Omar Khattab. 2025. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. *CoRR* abs/2507.19457 (2025).
- [4] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *NeurIPS*.
- [5] In Gim, Guojun Chen, Seung-Seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt Cache: Modular Attention Reuse for Low-Latency Inference. In *MLSys*.
- [6] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2024. DSPy: Compiling Declarative Language Model Calls into State-of-the-Art Pipelines. In *ICLR*.
- [7] Sunjun Kwon, Jiyoun Kim, Heeyoung Kwak, Dongchul Cha, Hanyul Yoon, Kwang Kim, Jeewon Yang, Seunghyun Won, and Edward Choi. 2024. EHRNoteQA: An LLM Benchmark for Real-World Clinical Practice Using Discharge Summaries. In *NeurIPS*.
- [8] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, and Gerardo Vitagliano. 2025. Palimpsest: Optimizing AI-Powered Analytics with Declarative Query Processing. In *CIDR*.
- [9] Harsha Nori, Yin Tat Lee, Sheng Zhang, Dean Carignan, Richard Edgar, Nicolò Fusi, Nicholas King, Jonathan Larson, Yuanzhi Li, Weishung Liu, Renqian Luo, Scott Mayer McKinney, Robert Osazuwa Ness, Hoifung Poon, Tao Qin, Naoto Usuyama, Christopher M. White, and Eric Horvitz. 2023. Can Generalist Foundation Models Outcompete Special-Purpose Tuning? Case Study in Medicine. *CoRR* abs/2311.16452 (2023).
- [10] Liana Patel, Siddharth Jha, Melissa Z. Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. 2025. Semantic Operators and Their Optimization: Towards AI-Based Data Analytics with Accuracy Guarantees in LOTUS. *PVLDB* 18, 11 (2025), 4171–4184.
- [11] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shrivani Agrawal, and Jeff Dean. 2023. Efficiently Scaling Transformer Inference. In *MLSys*.
- [12] Matthew Russo, Sivaprasad Sudhir, Gerardo Vitagliano, Chunwei Liu, Tim Kraska, Samuel Madden, and Michael J. Cafarella. 2025. Abacus: A Cost-Based Optimizer for Semantic Operator Systems. *CoRR* abs/2505.14661 (2025).
- [13] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G. Parameswaran, and Eugene Wu. 2025. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. *PVLDB* 18, 9 (2025), 3035–3048.
- [14] Shreya Shankar, Haotian Li, Parth Asawa, Madelon Hulsebos, Yiming Lin, J. D. Zambirscu-Pereira, Harrison Chase, Will Fu-Hinthorn, Aditya G. Parameswaran, and Eugene Wu. 2024. SPADE: Synthesizing Data Quality Assertions for Large Language Model Pipelines. *PVLDB* 17, 12 (2024), 4173–4186.
- [15] Arnav Singhvi, Manish Shetty, Shangyin Tan, Christopher Potts, Koushik Sen, Matei Zaharia, and Omar Khattab. 2023. DSPy Assertions: Computational Constraints for Self-Refining Language Model Pipelines. *CoRR* abs/2312.13382 (2023).
- [16] Sonish Sivarajkumar, Mark Kelley, Alyssa Samolyk-Mazzanti, Shyam Visweswaran, and Yanshan Wang. 2023. An Empirical Evaluation of Prompting Strategies for Large Language Models in Zero-Shot Clinical Natural Language Processing. *CoRR* abs/2309.08008 (2023).
- [17] Gemma Team. 2025. Gemma 3 Technical Report. *CoRR* abs/2503.19786 (2025).
- [18] Li Wang, Xi Chen, Xiangwen Deng, Hao Wen, Mingke You, Weizhi Liu, Qi Li, and Jian Li. 2024. Prompt engineering in consistency and reliability with the evidence-based guideline for LLMs. *npj Digit. Medicine* 7, 1 (2024).