Credo: Declarative Control of LLM Pipelines via Beliefs and Policies

Duo Lu
Brown University
duo_lu@brown.edu

Andrew Crotty
Northwestern University
andrew.crotty@northwestern.edu

Uğur Çetintemel
Brown University
ugur_cetintemel@brown.edu

ABSTRACT

AI assistants are becoming commonplace in domains that require stateful decision-making in continuously evolving environments. As such, correctness depends on an assistant's ability to flexibly adapt by incorporating new evidence or revising prior conclusions. However, existing approaches rely on either static control flow (e.g., fixed pipelines) or unreliable agentic reasoning.

We therefore introduce Credo, a database-backed semantic control plane for LLM pipelines that represents state as high-level *beliefs* and guides execution through declarative *policies* defined in terms of those beliefs. Through this design, Credo enables adaptivity based on runtime signals while also simplifying auditing and debugging. This demo will illustrate these concepts using an LLM pipeline built for a financial question answering benchmark.

PVLDB Reference Format:
Duo Lu, Andrew Crotty, and Uğur Çetintemel. Credo: Declarative Control of LLM Pipelines via Beliefs and Policies. PVLDB, 19(12): 4518 - 4521, 2026.
doi:10.14778/3827998.3828054

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/brown-db/credo.

1 INTRODUCTION

Imagine a financial analyst asking an AI assistant: "What was the year-over-year revenue change for Apple in Q3 2023?" To answer this question, the assistant must decide how much evidence to gather, which model to use for reasoning, and the best method for validating the result. Most LLM orchestration frameworks (e.g., LangChain [1]) involve static control flow, either in the form of a fixed pipeline or explicit instructions embedded directly in the prompts. This mixture of control flow with program logic makes the resulting behavior both rigid and difficult to reason about, especially when debugging. Moreover, developers must carefully balance the ability to handle more complex tasks against the risk of overprovisioning for simpler ones.

To illustrate this problem, we randomly selected 100 tasks from FinanceBench [4] and ran them with 18 different pipeline configurations representing all combinations of three Gemma 3 [12] model sizes, three prompting strategies, and two retrieval methods. The results, shown in Fig. 1, highlight two key findings.

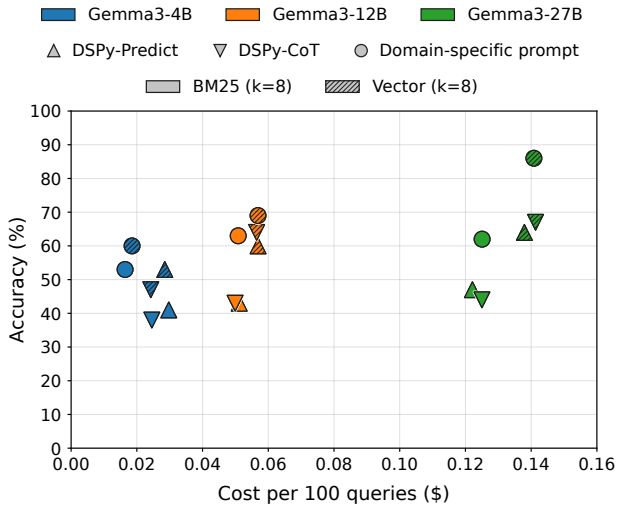

Figure 1: Accuracy vs. cost for each of the 18 configurations on 100 randomly chosen tasks from FinanceBench [4].

First, retrieval method matters more than model size. For example, with the 12B model and DSPy's [5] chain-of-thought (CoT) prompting strategy, simply switching the retrieval method from BM25 lexical search to vector similarity search increases accuracy from 43% to 64% at similar inference cost. More strikingly, the 4B model with vector retrieval outperforms the 27B model with BM25 retrieval, even though the latter has a 5× higher inference cost.

Second, tuning the prompting strategy alone provides clear improvements for virtually all combinations of model size and retrieval method. For instance, switching from DSPy-CoT to a domain-specific prompt with the 12B model and BM25 retrieval yields an accuracy improvement of 20 percentage points with no commensurate cost increase. In some cases (e.g., with the 4B model and BM25 retrieval), this prompting strategy can even improve accuracy while simultaneously reducing cost.

This complex space of potential optimizations motivates the need for an adaptive approach that can react to runtime signals and select the appropriate execution flow on a case-by-case basis. On one end of the spectrum are semantic query processing engines (e.g., LOTUS [10], DocETL [11], Palimpzest [7]), which expose composable LLM-powered operators for a more declarative approach to pipeline specification. However, control flow is typically fixed when lowering these pipelines to corresponding physical plans, perhaps with minor adaptivity baked into the prompts of individual operators. At the other extreme, agentic AI frameworks (e.g., ReAct [15], AutoGen [13]) have no predefined control flow path and instead rely entirely on LLM reasoning to drive execution. Agents offer substantial flexibility because they can reevaluate and modify

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828054

current plans based on the results of each execution step, yet this autonomy makes precise behavioral control and enforcement of certain guarantees extremely challenging.

Our approach, called Credo, achieves the best of both worlds by allowing developers to define logical pipelines composed of high-level operators, with adaptive physical execution decisions handled separately by a database-backed semantic control plane. Credo's core abstractions include *beliefs*, which represent claims about runtime state (e.g., query complexity, evidence relevance), and *policies*, which are declarative rules that prescribe how to adjust execution behavior in response to those beliefs. As new runtime signals emerge, Credo's execution engine reevaluates beliefs, triggers policies, and updates the physical plan accordingly. Consequently, these execution decisions become explicit, making runtime behavior more understandable and controllable. In this demo, the audience will interactively compose beliefs and policies to adaptively control an LLM pipeline built for FinanceBench.

2 CREDO

We argue that the logical definition of LLM pipelines (i.e., the types and ordering of operators) should be decoupled from physical execution decisions (e.g., model choice, evidence retrieval method, result verification) [2]. As shown in Fig. 2, Credo achieves this separation through three complementary layers: (1) an *observation layer* that makes runtime state explicit as high-level beliefs; (2) a *decision layer* that manages declarative policies over those beliefs; and (3) an *execution layer* that applies actions to adapt the pipeline when policies fire. In the following, we describe each in more detail.

2.1 Observation Layer: Beliefs

In the observation layer, Credo maintains a set of *beliefs* that describe the current state, expressed as `(value, confidence)` pairs representing semantic claims and any associated uncertainty. Beliefs are computed by extractors (e.g., heuristics, classifiers, LLM prompts) that either operate directly on raw evidence like documents or build on top of other beliefs. Credo can resolve static beliefs before execution begins, whereas dynamic beliefs depend on intermediate results.

Fig. 3 shows two example beliefs. The `query_complexity` belief wraps an implementation of RouteLLM [9], a lightweight LLM router that dynamically predicts whether a particular query will benefit from using a larger model. Note that this belief is static, meaning that Credo can perform the extraction based solely on information available prior to operator execution. The `relevance` belief, on the other hand, is dynamic, since it depends on information available only after retrieval (i.e., if retrieved documents are relevant for a particular query). This belief is implemented as an LLM-as-a-judge extractor that evaluates whether the documents actually contain the information necessary to answer the question, returning a `(value, confidence)` pair where `value` $\in$ {`correct`, `ambiguous`, `incorrect`}.

One obvious concern is the extraction cost, as some beliefs may be particularly expensive to evaluate (e.g., requiring LLM invocation). To mitigate this problem, we are developing a cost-based optimizer for belief extraction planning, such that expensive beliefs are extracted as infrequently as possible.

2.2 Decision Layer: Policies

The decision layer handles how and when to adapt the pipeline based on the current set of beliefs. Some existing frameworks support adaptivity in narrow circumstances, such as CRAG [14] for retrieval quality or Self-Refine [8] for answer quality. In contrast, Credo externalizes this logic into *policies*, which are declarative rules that map belief conditions to physical operator parameters. Each policy specifies target operators and a set of these physical parameters to apply, with trigger conditions that can consider both the belief values and their associated confidence scores. Developers can also assign an explicit priority to each policy, such that rewrites will occur in priority order if multiple policies fire at the same time. Much like declaring an index to change physical access paths without rewriting a SQL query, Credo enables policy tuning independent of the pipeline's control flow; that is, developers can extend the adaptivity of LLM pipelines simply by updating the policy catalog rather than the underlying implementation.

Fig. 3 shows two example policies. The first is an operator rewrite that fires when `query_complexity.value` $\geq$ `0.25` with sufficiently high confidence, modifying `generate` to use the largest available model. The second is a pipeline rewrite that fires when `relevance.value = incorrect`, indicating that the retrieved evidence does not support an answer to the question. The policy therefore invalidates existing state (i.e., `{evidence, model_answer, verified}`) and rewrites `retrieve` to use vector retrieval with more candidates. Note that these rewrites apply only to queries that trigger the policies, whereas those that do not will execute with the original pipeline.

2.3 Execution Layer: Actions

The execution layer applies the actions associated with policies whenever they fire and invokes their target operators. Consistent with Credo's separation between logical pipeline definition and physical execution, operators have no direct knowledge of beliefs or policies; they simply receive a set of parameters and execute. Instead, the semantic control plane handles all planning and optimization, similar to the traditional separation between logical relational operators (e.g., joins) and selection of a physical implementation (e.g., hash vs. sort-merge).

Credo currently provides three built-in operators: (1) `retrieve`, which gathers evidence using a configurable document retrieval strategy; (2) `generate`, which invokes the chosen LLM with a customizable prompting strategy; and (3) `verify`, which evaluates answer correctness via techniques like LLM-as-a-judge or chain-of-verification [3]. Developers can also register user-defined operators for their applications (e.g., custom API calls, domain-specific tools). In the future, we plan to extend Credo's built-in operator set to include functionality like web search, code interpreters, and storage for long-term agentic memory.

3 DEMO DESCRIPTION

The demo will center on FinanceBench [4], a financial question answering benchmark with tasks ranging in difficulty from simple calculations to multi-step, cross-document reasoning. The audience will use Credo's web interface (Fig. 4) to create custom beliefs and policies while observing their effects on pipeline execution.

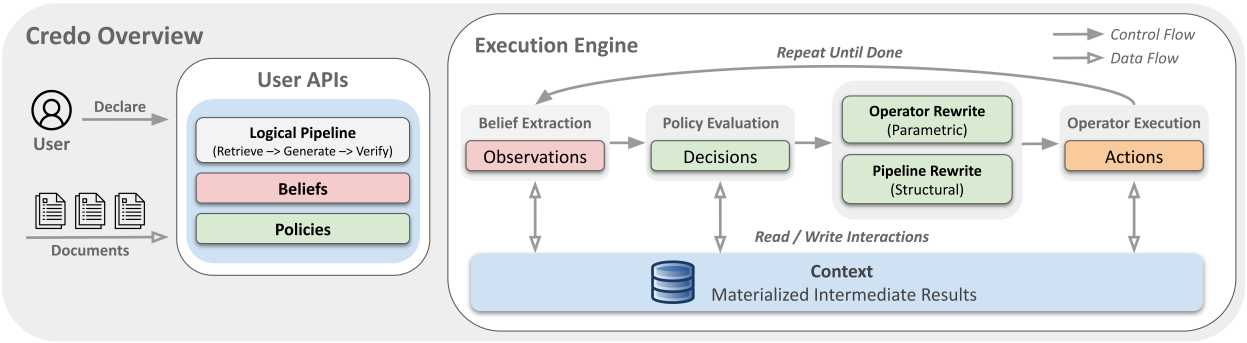

Figure 2: Overview of Credo. Developers specify a logical LLM pipeline accompanied by a set of declarative beliefs and policies that form its semantic control plane, allowing adaptivity without directly modifying the underlying code. The observation layer evaluates beliefs based on available evidence, the decision layer evaluates policies over belief values and associated confidence scores, and the execution layer applies the corresponding rewrites before continuing with the updated physical plan.

```
/* static */
BELIEF query_complexity(input:query):
    (value, confidence) = routellm_router(input:query)

/* dynamic */
BELIEF relevance(input:query, context:evidence):
    /* value ∈ {correct, ambiguous, incorrect} */
    (value, confidence) =
            llm_judge(input:query, context:evidence)
    DEPENDS ON {context:evidence}

/* operator rewrite */
WHEN query_complexity.value >= 0.25
     AND query_complexity.confidence >= 0.7
THEN REWRITE generate SET model = gemma3-27b

/* pipeline rewrite */
WHEN relevance.value = incorrect
     AND relevance.confidence >= 0.8
THEN INVALIDATE {evidence, model_answer, verified}
     REWRITE retrieve SET method = vector, top_k = 12
```

Figure 3: Example belief and policy definitions.

As a jumping-off point, we will begin with two tasks from FinanceBench that appear superficially similar but demand very different execution strategies. The first is a simple lookup that requires locating a single numerical value in an income statement: "What was Apple's total revenue in FY2022?" The second requires extracting two different numerical values from a table and then performing a two-step computation: "What is Microsoft's FY2022 gross profit margin?" As explained, a fixed pipeline would either waste resources on the former or risk errors on the latter.

Using these two questions, the audience will receive a brief tutorial that illustrates how Credo solves this problem through our framework of beliefs and policies. First, we will register the RouteLLM [9] router from Fig. 3 as a belief, as well as specify corresponding retrieval and model choice policies (e.g., use the 4B model and BM25 retrieval with k=6 below a 0.12 threshold). We will also define a belief modeled on CRAG to evaluate whether the retrieved evidence is actually relevant to the question and related policy to switch to vector retrieval whenever this belief returns `incorrect`. A similar `complex_calculation` belief will determine if an answer requires combining multiple pieces of evidence, with a policy to regenerate using the 27B model and a modified prompt describing this requirement. Finally, the `groundedness` and `has_hallucinations` beliefs will assess whether the gathered evidence fails to support the conclusions and detect any potential hallucinations in the answer, respectively, either of which will trigger another corrective policy to retry the generation.

After the tutorial, the audience can freely explore Credo by submitting predefined tasks from the benchmark or their own custom queries via the query box ①. The logical LLM pipeline consists of three steps (`retrieve` → `generate` → `verify`) and appears just below the query box ②. The interface also displays the current belief catalog ③ and policy catalog ④, which can be adjusted using the inline editor to see how changes affect the compiled plan. Clicking *Start* will compile and execute the pipeline, evaluating beliefs throughout execution and updating the physical plan as policies fire ⑤. When the task completes, the interface will show the final answer along with details about every execution step ⑥, including a cost breakdown for each operator, trace of all belief evaluations and triggered policies, and summary of corrective rewrites applied.

4 CONCLUSION & FUTURE WORK

We presented Credo, a framework for controlling the runtime behavior of LLM pipelines through declarative beliefs and policies. During the demonstration, the audience will observe firsthand how Credo can support runtime adaptivity for these pipelines without changing the underlying application code, all while making execution decisions more transparent.

Moving forward, we plan to extend Credo with new operators and tools to support a broader range of applications. Our longer-term goal is to enable richer agentic workflows, potentially even automating the discovery of new beliefs and policies. For example, a search process could propose candidates, test them against execution outcomes, and retain successful strategies for reuse across tasks, similar to approaches like Meta-Harness [6]. The resulting optimization loop would substantially reduce the burden on developers while keeping these decisions explicit.

ACKNOWLEDGMENTS

This work was supported in part by a Brown Seed Grant and an award from Google's Research Scholar Program.

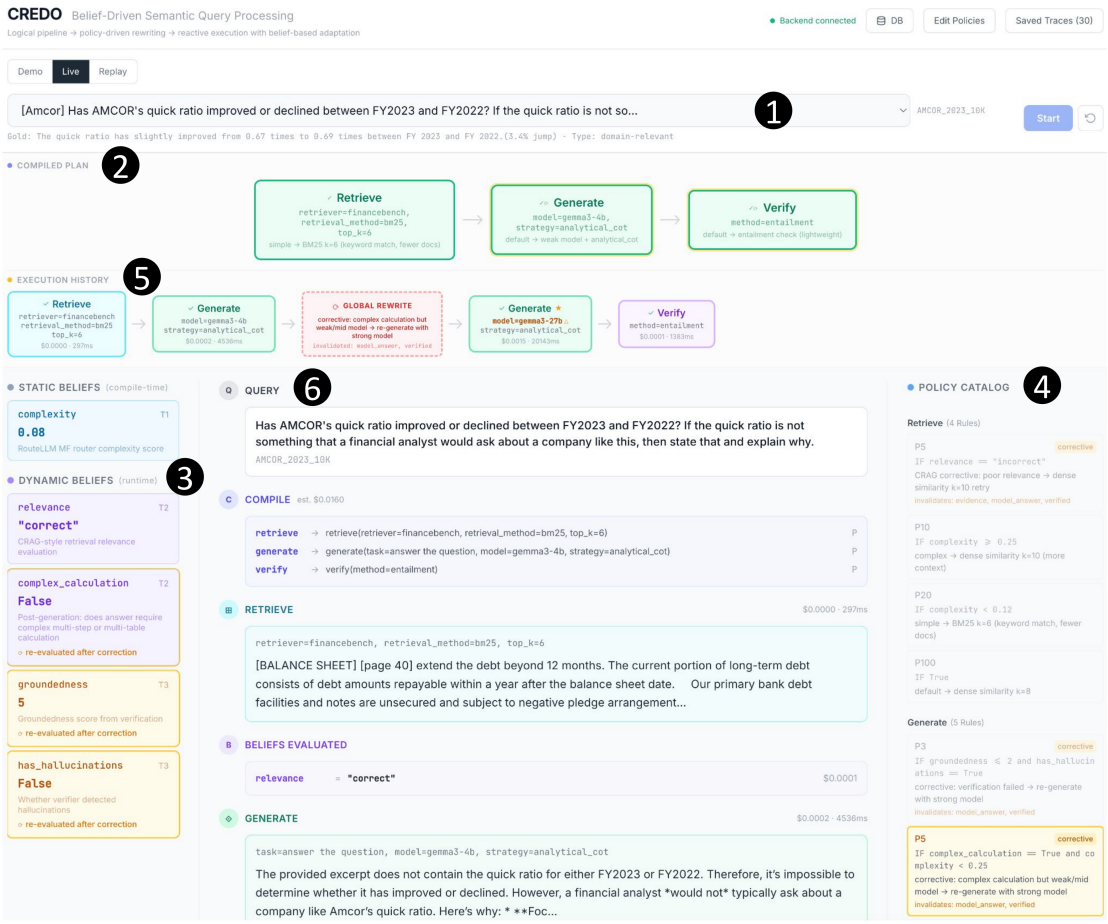

Figure 4: CREDO's web interface. ① *Query Box:* Allows selecting from predefined benchmark tasks or submitting a custom query. ② *Logical Plan:* Shows the high-level operators that make up the LLM pipeline. ③ *Belief Catalog:* Reports each belief's current (`value`, `confidence`) pair based on the available evidence and allows for the definition of new beliefs. ④ *Policy Catalog:* Lists the active policies and supports inline updates, as well as the creation of new policies. ⑤ *Physical Plan:* Displays the operator configurations produced during compilation and highlights subsequent rewrites as policies fire. ⑥ *Event History:* Records a trace of evaluated beliefs, triggered policies, and operator executions, including details like inputs/outputs, cost, and latency.

REFERENCES

[1] 2026. LangChain. https://www.langchain.com/.

[2] Ugur Çetintemel, Shu Chen, Alexander W. Lee, Deepti Raghavan, Duo Lu, and Andrew Crotty. 2026. Making Prompts First-Class Citizens for Adaptive LLM Pipelines. In *CIDR*.

[3] Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. 2024. Chain-of-Verification Reduces Hallucination in Large Language Models. In *ACL (Findings)*. 3563–3578.

[4] Pranab Islam, Anand Kannappan, Douwe Kiela, Rebecca Qian, Nino Scherrer, and Bertie Vidgen. 2023. FinanceBench: A New Benchmark for Financial Question Answering. *CoRR* abs/2311.11944 (2023).

[5] 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*.

[6] Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. 2026. Meta-Harness: End-to-End Optimization of Model Harnesses. *CoRR* abs/2603.28052 (2026).

[7] Chunwei Liu, Matthew Russo, Michael J. Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael J. Franklin, Tim Kraska, Samuel Madden, Rana Shahout, and Gerardo Vitagliano. 2025. Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing. In *CIDR*.

[8] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. In *NeurIPS*.

[9] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M. Waleed Kadous, and Ion Stoica. 2025. RouteLLM: Learning to Route LLMs from Preference Data. In *ICLR*.

[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. *PVLDB* 18, 11 (2025), 4171–4184.

[11] 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.

[12] Gemma Team. 2025. Gemma 3 Technical Report. *CoRR* abs/2503.19786 (2025).

[13] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation Framework. *CoRR* abs/2308.08155 (2023).

[14] Shi-Qi Yan, Jia-Chen Gu, Yun Zhu, and Zhen-Hua Ling. 2024. Corrective Retrieval Augmented Generation. *CoRR* abs/2401.15884 (2024).

[15] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *ICLR*.