Building Reactive Large Language Model Pipelines with Motion

Shreya Shankar
UC Berkeley
shreyashankar@berkeley.edu

Aditya G. Parameswaran
UC Berkeley
adityagp@berkeley.edu

ABSTRACT

Large language models (LLMs) rely on prompts with detailed and informative context to produce high-quality responses at scale. One way to develop such prompts is through *reactive* LLM pipelines, which incorporate new information—e.g., end-user feedback and summaries of historical inputs and outputs—back into prompts to improve future response quality. We present Motion, a Python framework to build and execute reactive LLM pipelines. Motion uses a weak consistency model to maintain prompt versions, trading off freshness for end-user latency. We demonstrate Motion with an e-commerce application that suggests apparel to wear for any event, allowing attendees to indirectly influence prompts with their queries. Attendees can interact with the demo as end-users or modify the application as developers, adding new information sources for reactive prompts.

CCS CONCEPTS

• **Information systems** → **Data management systems**; • **Computing methodologies** → **Artificial intelligence**.

KEYWORDS

Large language models, incremental view maintenance

ACM Reference Format:
Shreya Shankar and Aditya G. Parameswaran. 2024. Building Reactive Large Language Model Pipelines with Motion. In *Companion of the 2024 International Conference on Management of Data (SIGMOD-Companion '24), June 9–15, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3626246.3654734

1 INTRODUCTION

Large language models (LLMs) are increasingly used in intelligent software and data pipelines. Essential to these pipelines is the *prompt*, or a string input to an LLM. Due to in-context learning, LLM responses for domain-specific tasks improve when prompts include external information, example inputs-and-outputs, and even LLM-synthesized instructions [1, 4–6]. In production LLM pipelines, any new information—be it user feedback, external data, or the initial input to the pipeline—should inform future prompts; as such, we call them *reactive* prompts. For instance, Figure 1 shows an e-commerce-focused pipeline that generates a personalized note suggesting clothing to buy for an event. Its prompt could change over time by including LLM-generated insights from user interactions. For a concrete example of this, consider the following sequence of user interactions, where 1, 3, and 4 show prompts for event styling queries and 2 corresponds to user feedback:

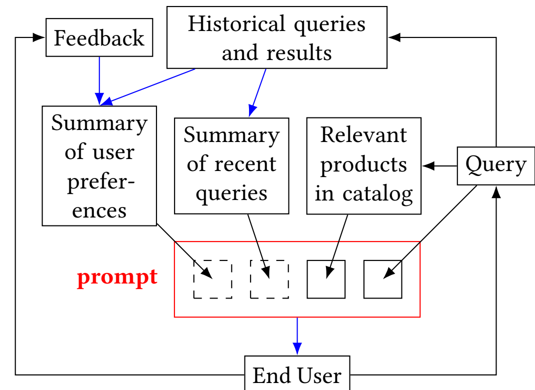


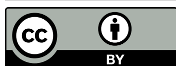
Figure 1: Example of a reactive prompt for an LLM pipeline tailored for e-commerce, which recommends new products to buy for an event query. Blue arrows represent LLM-enabled data transformations. Dashed prompt sub-parts do not always have to be up-to-date—in fact, requiring these to be fresh can have latencies of over half a minute when there are multiple historical queries and feedbacks.

(1) "What apparel items should I buy for SIGMOD in Chile?"
(2) *User disliked "purple blazer" and liked "wide-leg jeans."*
(3) "I work in tech. I dress casually. What apparel items should I buy for hiking in the Bay Area?"
(4) "I work in tech and have an active lifestyle. I dress casually. What apparel items should I buy for coffee with a friend?"

In the above sequence, blue phrases in the prompt are dynamically generated based on previous user-generated activity. The new context can improve the quality of responses.

Maintaining Reactive Prompts in Production is Hard. New information to incorporate into the prompt is typically unstructured and grows in size over time. Thus, LLMs must summarize or intelligently extract relevant data to include in prompt(s), which can be too costly or time-consuming for production settings [3]. For example, in Figure 1, if there are a handful of historical queries and feedbacks, GPT-4 can take more than thirty seconds to generate a summary [2]. Waiting on a reactive prompt to be fully up-to-date with the latest information can be unacceptable. Second, there are significant engineering efforts required to maintain reactive prompts. Maintaining code to glue together online and offline (cached) data for ML pipelines is a pain point for practitioners [8, 9].

Reactive Prompts are Views. To maintain reactive prompts, we conceptualize prompts as *views* over unstructured information. Each new piece of information creates a new version of the prompt, so our problem is an instance of incremental view maintenance



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

SIGMOD-Companion '24, June 9–15, 2024, Santiago, AA, Chile
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0422-2/24/06
https://doi.org/10.1145/3626246.3654734

(IVM). While IVM techniques have been used to optimize ML pipelines before, e.g., [7], new considerations arise when LLMs generate view results. First, prompts need not include all new information. The dashed boxes in Figure 1 represent context that could be stale without impacting overall correctness, even if result quality may be slightly diminished. For example, while the summary of historical queries may improve future recommendations because it provides more context on the end-user, it does not have to reflect all queries. This motivates a *weak consistency model* for reactive prompts: a system can maintain stale versions of prompt sub-parts to quickly support LLM pipeline queries at scale. Second, even in variants of IVM that admit staleness by batching updates [10], materialized views are typically derived from a single snapshot of the underlying data. However, here, prompt sub-parts may vary in version: in our example, different end-users could experience varying staleness levels in query summaries based on when and how they access the LLM pipeline. We might batch summarization requests for different users around anticipated end-user visits, posing a scheduling problem. Moreover, there is a tradeoff between summarizing small batches of history and concatenating the summaries, versus generating a summary based on the entire history. Overall, updating and managing state across many interacting components in the system is a huge challenge.

Supporting Reactive Prompts with Motion. We introduce Motion, a Python-based framework for creating and executing LLM pipelines with reactive prompts, using a weak consistency model. Motion decouples the prompt from the end-to-end LLM pipeline, maintaining versions of prompt sub-parts to be read anytime. Designed for flexibility, Motion gives developers control over all LLM prompts and allows them to define how to incrementally update prompt sub-parts. Designed for scalability, Motion operates in serverless and cloud settings, with versions of prompt sub-parts saved in a key-value store (Redis in our implementation).

Demonstration Scenario. In a real-world application, Motion is actively used in a fashion startup, powering various LLM pipelines that require domain-specific knowledge (e.g., recommending clothing items to users based on their query history and feedback on previous recommendations). In our demo, we will show a simplified example of an e-commerce application to suggest apparel items to buy. This pipeline maintains reactive prompts based on a user's query history. We describe the demo in Section 3, and the code for the demo can be found on Github.[1] Motion is also open-source.[2]

2 SYSTEM

Motion is written in Python. Developers define Python functions for computing prompt sub-parts, distinguishing between those for real-time computation and those processed in the background with permissible staleness. Motion manages execution of these UDFs, reading and writing sub-parts to a key-value store.

2.1 Motion Primitives and Interface

In Motion, the primary abstraction is the *Component*, which includes state (prompt sub-parts) and operations for state manipulation. State is a Python dictionary, initialized by a developer-defined default. Developers manage this state via *flows*, each consisting of zero or one real-time *serve* operations (read-only) and zero or more asynchronous *update* operations that write new values to the state. Serve operations handle real-time LLM pipeline logic, combining runtime arguments with potentially-stale state, while update operations recompute values in the state. When a serve operation finishes, the result is immediately returned, and any update operations run in the background. Components can support multiple flows, triggered by various events like human feedback, and can be serve-only or update-only. Flows only have one serve operation but can include several update operations, which is useful when one event (e.g., user feedback) needs to update multiple prompt sub-parts. Serve and update operations are defined as Python functions, instrumented with decorators.

Writing Figure 1 as a Motion Component. To write the pipeline in Figure 1 as a Motion component, we first identify the prompt sub-parts to maintain and set up some initial values for them:

```
from motion import Component

FashionPrompt = Component("Fashion")

@FashionPrompt.init_state
def setup():
  return {"query_summary": "No queries yet.", "preference_summary": "No
  ↪ preference information yet."}
```

Next, we write the flow `styling_query`. The corresponding serve operation assembles the LLM prompt (red rectangle in Figure 1) and calls the LLM, returning the LLM's response:

```
@FashionPrompt.serve("styling_query")
def generate_recs(state, props):
    # Props = properties to this specific flow's execution
    # First retrieve products from the catalog
    catalog_products = retrieve(props['event'])
    prompt = f"Consider the following lifestyle and preference information
    ↪ about me: {state['query_summary']}, {state['preference_summary']}.
    ↪ Suggest 3-5 apparel items for me to buy for {props['event']}, using
    ↪ the catalog: {catalog_products}."
    return llm(prompt)
```

Note that the `props` argument is a dictionary of key-value pairs passed into the flow at runtime (e.g., the event to be styled for). The `styling_query` flow would also have an update operation to incrementally maintain the summary of recent queries:

```
@FashionPrompt.update("styling_query")
def query_summary(state, props):
    # props.serve_result contains the result from the serve op
    prompt = f"You recommended a user buy {props.serve_result} for {props['
    ↪ event']}. The information we currently have about them is: {state['
    ↪ query_summary']}. Based on their query history, give a new 3-
    ↪ sentence summary about their lifestyle."
    query_summary = llm(prompt)
    # Update state
    return {"query_summary": query_summary}
```

In the above code snippet, the update operation runs in the background after the serve result is returned to the end-user. We can also define a flow, `feedback`, with only an update operation to maintain the state's `preference_summary`, but we omit this code for brevity. To run the `styling_query` flow for a specific event, the code would look like:

```
if __name__ == "__main__":
    instance = FashionPrompt(user_id) # Some user_id
    instance.run("styling_query", props={"event": "sightseeing in Santiago,
    ↪ Chile"})
```

[1]https://github.com/shreyashankar/motion-sigmod-demo
[2]https://github.com/dm4ml/motion

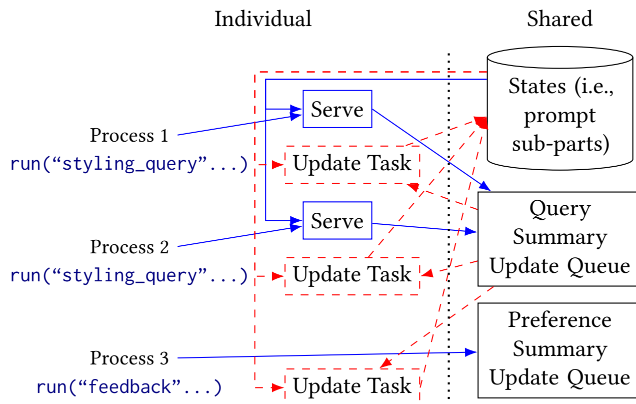


Figure 2: Diagram of Motion's execution model with 3 concurrently-running processes for the same component instance. The instance corresponds to the pipeline in Figure 1. Blue serve operations run concurrently in real-time, assembling prompts from potentially-stale state. Red dashed operations run in the background and run one-at-a-time (since update operations require an exclusive lock). States and queues are shared across all processes.

States are stored in Redis, so even when the instance goes out of scope, the state persists. Initial state is derived from the `init_state` function. Persisting state like this for developers allows them to easily run Motion components in the cloud and even serverless environments. The startup that uses Motion executes flows in serverless functions triggered by their web app, where the function code looks like the following:

```
from my_components import FashionPrompt

with FashionPrompt(user_id, flush_on_exit=True) as instance:
    # flush_on_exit will process user_id's ``styling_query'' update queue
    # on context manager teardown
    yield instance.run("styling_query", props={"event": "sightseeing in
    ↪ Santiago, Chile"})
```

2.2 Motion's Execution Model

For scalability, any number of Python processes can connect to the same Motion component instance and concurrently run flows. Motion ensures that in this distributed setting, component instance states are weakly consistent.

Component Instance Creation. When a component instance object is created (i.e., `instance = FashionPrompt(id)`), Motion creates a background thread (or process if preferred by the developer) to execute any update operations. Figure 2 shows that this background task subscribes to a Redis-backed queue for each update operation. Each component instance id has its own set of queues.

Executing Flows: Serve Operations. When a developer runs a flow for the instance (i.e., `instance.run(styling_query,props={k:v})`), the serve operation corresponding to `styling_query` gets executed, as shown in Figure 2. Motion achieves weak consistency by maintaining some version of the state, which may be stale, to be read at any time. At a high level, executing a serve operation involves running the relevant Python function with a potentially-stale state, sending the result to a queue to be processed in any background operations, and returning the result to the end-user. More concretely, executing a serve operation does the following:

(1) Identify the Python function f decorated with `@C.serve(styling_query)`
(2) Execute f to get result r, using the latest `state` for this component instance ID from Redis (or the developer-defined initial state if no state exists) and `{k: v}` for `props`
(3) Add r and `props` to every queue linked to Python functions decorated with `@C.update(styling_query)`
(4) Return r to the user

Since serve operations are not allowed to modify state, they can be executed concurrently for minimal end-user latency.

Executing Flows: Update Operations. Since any number of Python processes can create objects with the same component instance id, and each process has its own background thread or process to process update operations off update queues (Figure 2), state updates can get lost or overwrite each other if we don't isolate them. As a simple solution, Motion requires each update operation to hold a lock exclusive to the component instance while executing and updating state. However, the drawback is that updates cannot execute concurrently, so if update operations finish slower than the rate at which flows with update operations are executed, there can be significant backpressure. The current solution to handle backpressure discards unprocessed update operations older than a developer-defined period of time (i.e., n seconds) or number of subsequent new update operations (i.e., if there are n newer update operations that haven't been processed yet). We are exploring different solutions to process update operations in parallel, e.g., batch processing with LLMs, an optimistic concurrency control approach, which may relieve the backpressure as a side effect.

3 DEMONSTRATION SCENARIO

Our demonstration[3] will allow attendees to both interact with a Motion component as an end-user, by running LLM pipelines and observing prompts "learn" from their interactions, as well as pretend to be a developer, by adding a new flow to create new prompt sub-parts. We demonstrate Motion with an example e-commerce application that generates personalized recommendations and notes for what to wear to an event (e.g., ski trip, conference). The demo will be presented via a Streamlit dashboard running on our laptop, where the left half shows the Motion application, and the right half shows prompt internals. Attendees will interact with this application by submitting event styling queries (e.g., "what should I wear to a ski trip?") and viewing the results, as well as watching prompt sub-parts update independently from the event styling queries on the right side of the dashboard.

Demo Component Architecture. The demo's 250-line Python code primarily consists of prompt string templates and LLM function calling. It features one component with two flows: `styling_query` for outfit recommendations and `note` for personalized notes, both using GPT-3.5-Turbo. The `styling_query` serve operation suggests up to five apparel items items to buy for an event and retrieves images (using the Google Images API), while `note` creates a 3-sentence

[3] https://github.com/shreyashankar/motion-sigmod-demo

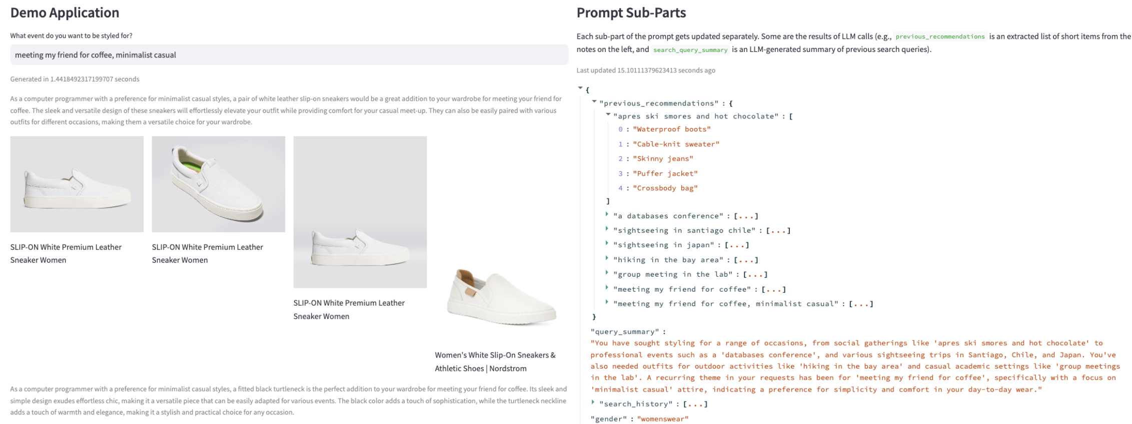


Figure 3: Screenshot of our demonstration. The left-hand side shows an example e-commerce application to recommend apparel to wear for a user-specified event, where the recommendations and personalized notes are generated in 1.5 seconds. The right-hand side displays component state (i.e., prompt sub-parts) as they update in the background.

note for why the end-user should buy some apparel item. The component state has some static prompt sub-parts like age, gender, and occupation. Two dynamic sub-parts, updated using GPT-4, are `previous_recs` (a dictionary of past queries and LLM-extracted commonly-suggested apparel items to avoid recommending again) and `query_summary` (LLM-generated user lifestyle and wardrobe preference insights). Both sub-parts are reactively updated in the background from update operations triggered by `styling_query`.

Interactive Demonstration. Attendees can play two roles in the demonstration: the end-user and the LLM pipeline developer. As an end-user, first, attendees will submit basic demographic information to initialize static prompt sub-parts, like age and gender. Then, in the Streamlit dashboard, attendees will submit several event styling queries of their choice (e.g., "SIGMOD conference" or "sightseeing in Santiago, Chile"). For every query, attendees will observe apparel suggestions arrive within a couple of seconds, and they will notice the notes become more personalized as they submit more queries (due to the `query_summary` prompt sub-part). If they submit the same query, they will notice different suggestions (due to the `previous_recs` prompt sub-part). Then, as a developer, in the Streamlit dashboard, attendees will observe prompt sub-parts changing over time as the corresponding update operations finish executing. We guide the attendee to improve the LLM pipeline in Motion by adding a new flow to learn from feedback on apparel recommendations. We will add a new prompt sub-part: `item_feedback`. We will write a new flow with only an update operation, which uses an LLM to process the feedback (i.e., thumbs-up or thumbs-down and optional text feedback) and generate the `item_feedback` sub-part. Then, we will add `item_feedback` to the string prompt submitted to the LLM in the real-time serve operations. Once we have finished modifying the code, we will redeploy the Motion component and experiment with this new functionality by liking and disliking several items and watching the recommendations adapt to our feedback. Finally, we will show attendees our Component Visualization tool[4], which helps LLM pipeline developers view the dataflows in complex MOTION components.

Acknowledgements. We acknowledge the support from grants DGE2243822, IIS-2129008, IIS-1940759, and IIS-1940757 awarded by the National Science Foundation, an NDSEG Fellowship, funds from the Alfred P. Sloan Foundation, as well as EPIC lab sponsors: GResearch, Adobe, Microsoft, Google, and Sigma Computing.

REFERENCES

[1] Mahyar Abbasian, Iman Azimi, Amir M. Rahmani, and Ramesh Jain. 2024. Conversational Health Agents: A Personalized LLM-Powered Agent Framework. arXiv:2310.02374 [cs.CL]
[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).
[3] Phillip Carter. 2023. All the hard stuff nobody talks about when building products with llms. https://www.honeycomb.io/blog/hard-stuff-nobody-talks-about-llm
[4] Zui Chen, Zihui Gu, Lei Cao, Ju Fan, Sam Madden, and Nan Tang. 2023. Symphony: Towards natural language query answering over multi-modal data lakes. In *Conference on Innovative Data Systems Research, CIDR*. 8–151.
[5] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey for in-context learning. *arXiv preprint arXiv:2301.00234* (2022).
[6] Jerry Liu. 2023. Data Agents. https://blog.llamaindex.ai/data-agents-eed797d7972f
[7] Milos Nikolic, Mohammed ElSeidy, and Christoph Koch. 2014. LINVIEW: incremental view maintenance for complex analytical queries. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 253–264. https://doi.org/10.1145/2588555.2610519
[8] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. *Advances in neural information processing systems* 28 (2015).
[9] Shreya Shankar, Rolando Garcia, Joseph M Hellerstein, and Aditya G Parameswaran. 2022. Operationalizing machine learning: An interview study. *arXiv preprint arXiv:2209.09125* (2022).
[10] Dixin Tang, Zechao Shang, Aaron J Elmore, Sanjay Krishnan, and Michael J Franklin. 2019. Intermittent query processing. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1427–1441.

[4]https://dm4ml.github.io/motion-vis/