Structured State Management for Agentic Data Pipelines: Bounded Context, Schema Contracts, and Selective Recovery

Shubhi Asthana
IBM Research
San Jose, California, USA
sasthan@us.ibm.com

Chad DeLuca
IBM Research
San Jose, California, USA
delucac@us.ibm.com

Hima Patel
IBM Research
Bengaluru, Karnataka, India
himapatel@in.ibm.com

ABSTRACT

Agentic systems operating over heterogeneous data ecosystems face data management challenges distinct from single-turn LLM inference. When a pipeline ingests large artifacts—log streams, database query results, document corpora—into a monolithic context window, the result is unbounded, non-queryable state that is expensive to recover on failure.

We present *structured state management* (SSM), a data-centric architectural pattern in which each pipeline stage reads only its declared upstream keys, every stage output is schema-validated before commit to a persistent state store, and the execution engine performs selective recovery—retrying only the failed stage over its bounded context.

We evaluate SSM on two data-intensive workloads: Kubernetes root cause analysis (84,000-record input artifact) and multi-file code debugging. Across three configurations (monolithic, static-decomposed, SSM), we find that static decomposition *increases* retry cost by 80.5% over monolithic (1,632 ± 145 vs. 904 ± 17 tokens) due to cascading re-reads of upstream state, while SSM reduces retry cost by 73.2% over static decomposition and 51.7% over monolithic (436 ± 132 tokens). Schema contracts also create structured human-in-the-loop intervention points at every stage boundary.

VLDB Workshop Reference Format:
Shubhi Asthana, Chad DeLuca, and Hima Patel. Structured State Management for Agentic Data Pipelines. VLDB 2026 Workshop: DASHSys: Systems for Data-centric Agents with Human-in-the-loop.

1 INTRODUCTION

LLM-powered agents are increasingly deployed as *data pipelines*: they ingest structured logs, database results, knowledge graph traversals, and document corpora; classify, aggregate, and reason over those inputs; and produce structured outputs feeding downstream systems. Site reliability agents process thousands of metric time-series per incident [3]; code review agents ingest full repository histories; data quality agents scan entire table columns. These workloads create data management challenges—bounding ingestion cost, validating intermediate results, recovering from partial failures—that are recognisably the same problems that batch and streaming pipeline research has addressed for decades.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

Current agent frameworks treat these challenges as afterthoughts. AutoGen [20], LangGraph [8], and DSPy [6] typically pass full data artifacts as LLM context, producing three failure modes:

- **Unbounded ingestion cost.** Every LLM call re-reads the full input regardless of which portion is relevant.
- **Non-queryable intermediate state.** Results are embedded as unstructured text in subsequent prompts rather than typed, addressable records.
- **Cascading recovery.** On failure, the pipeline re-reads and re-executes all downstream steps from the failure point, paying full ingestion cost repeatedly.

These are data management failures, not model failures: the absence of schemas, typed stage interfaces, persistent queryable state, and dependency-aware recovery.

Our contribution. We present *structured state management* (SSM), which organises an agentic pipeline as a sequence of *judgment stages*, each with declared input and output schemas. Stage outputs are committed to a *persistent state store* as typed records keyed by stage identifier. Downstream stages read only their declared dependencies—a relational projection over pipeline state rather than full context re-ingestion. Failed stages are retried over their bounded input only.

The central empirical finding is counterintuitive: *static decomposition increases recovery cost over monolithic* (1,632 ± 145 vs. 904 ± 17 tokens, +80.5%), because sequential pipelines without runtime branching must re-execute all downstream stages from the failure point. SSM reduces recovery cost to 436 ± 132 tokens—a 51.7% reduction over monolithic and a 73.2% reduction over static.

Our contributions are: (1) a data-centric formalisation of SSM, connecting agentic pipeline design to established data management primitives (§2); (2) a three-configuration empirical evaluation showing that runtime-controlled recovery, not decomposition structure alone, drives efficiency gains (§3); and (3) a discussion of schema contracts as a structured human-in-the-loop mechanism (§4).

2 STRUCTURED STATE MANAGEMENT

SSM organises a pipeline into three interacting components (Figure 1).

Judgment Stages. Each stage s declares input dependencies D_s (the state-store keys it reads) and output schema Σ_s. Its context is exactly $\bigcup_{k \in D_s} \text{StateStore}[k]$—a relational projection over accumulated pipeline state. This enforces *context minimality* and *schema integrity* by construction.

State Store. A persistent, append-only key-value store keyed by stage identifier. Downstream stages access upstream results by

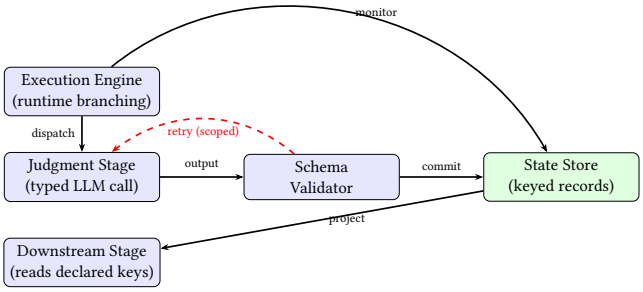

Figure 1: SSM architecture. Judgment stages are dispatched over projected state; outputs are schema-validated before commit. Failed stages are retried over their bounded context only.

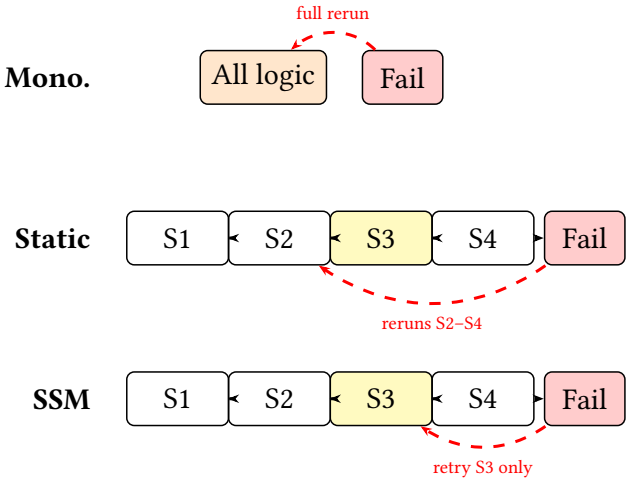

Figure 2: Recovery behavior under stage failure. Monolithic reruns the full pipeline; static reruns all downstream stages; SSM retries only the failed stage.

key, not by position in a context buffer, enabling non-sequential dependency arcs. In the RCA pipeline (§3), the Log Triage stage feeds both Anomaly Classification and Root Cause Analysis directly—a skip arc impossible in purely sequential context accumulation. This addresses the bounded-context problem identified in prior work [12, 16]: rather than engineering around context limits, SSM eliminates unnecessary re-ingestion structurally.

Execution Engine and Recovery. At each stage: (1) project D_s from the state store; (2) invoke the LLM call; (3) validate output against Σ_s; (4) on pass, commit to the state store; (5) on fail, retry over the same projected context, appending the validation error. If all retries fail, raise a structured exception. Figure 2 illustrates the recovery difference. The retry scope is exactly the failed stage over its declared dependencies—no upstream data is re-ingested, no downstream stages are re-executed.

Implementation. We use the Mellea framework [15], which provides typed LLM calls with enforced input/output schemas, persistent session state, and automated validation-and-repair:

```
result = session.instruct(
  prompt = "Classify anomalies from triage JSON."
           " Output: [{anomaly,type,severity,conf}]",
  requirements = [
    "Output must be a JSON list.",
    "Each item must have: anomaly, type,
     severity, confidence."
  ])
# On failure, Mellea appends the validation
# error and retries with scoped context only.
```

3 EMPIRICAL EVALUATION

Configurations. (i) **Monolithic**: full task logic in a single LLM prompt; (ii) **Static**: same stage graph as SSM, fixed sequential execution, no runtime branching; (iii) **SSM**: our approach with schema-validated state store and selective stage recovery. The static baseline isolates whether efficiency gains come from decomposition structure alone or from runtime recovery.

All configurations use `gpt-4` at temperature 0; tokens are measured via `tiktoken`; each is run **10 times**. Non-zero standard deviations at temperature 0 reflect API-side scheduling variability, not sampling randomness. Recovery cost is measured under controlled failure injection (one missing required JSON field), since natural failure rates are low at temperature 0; controlled injection enables statistically stable comparison without thousands of pipeline runs.

Natural failure rates. Natural schema validation failures occurred in 2 of 100 stage executions in Workload 1 (2.0%) and 0 of 40 in Workload 2, confirming that the primary robustness concern is recovery cost architecture, not failure frequency.

3.1 Workload 1: Kubernetes Root Cause Analysis

The task involves an OOMKill on `payment-svc` (deployment v2.4.2), database connection pool exhaustion, and a p99 latency SLA breach—ground truth: a deployment regression introducing unbounded in-memory batch processing. The input artifact totals 84,000 invoice records across raw container logs, metric time-series, and deployment event streams.

The SSM pipeline uses five stages: Log Triage (reads raw logs), Anomaly Classification (reads triage JSON), Root Cause Analysis (reads triage JSON + anomaly list via skip arc), Remediation Planning (reads RCA JSON), and Report Synthesis (reads RCA + remediation). No stage re-ingests the raw log artifact after Stage 1. The static baseline uses the same five stages sequentially; on Stage 3 failure, it reruns Stages 3–5.

Static decomposition *increases* recovery cost by 80.5% over monolithic (1,632 vs. 904 tokens): sequential execution must re-run Stages 3–5, whose combined cost exceeds the monolithic single call. SSM retries only Stage 3 over its declared dependencies, consuming 436 ± 132 tokens—a **51.7% reduction** over monolithic and **73.2% reduction** over static. Framework overhead (5.18s, ≈18% of total latency) is dominated by LLM API cost. SSM's higher normal-execution token count (2,716 vs. 904 for monolithic) reflects five

Table 1: Kubernetes RCA across 10 runs. Recovery cost under simulated Stage 3 failure.

Metric	Mono.	Static	SSM
Tokens (mean ± sd)	904 ± 17	2,553 ± 224	2,716 ± 424
Latency s (mean ± sd)	10.41 ± 2.31	22.76 ± 2.52	28.78 ± 5.06
LLM API (s)	10.41	18.66 ± 2.07	23.60 ± 4.15
Framework (s)	—	4.10 ± 0.45	5.18 ± 0.91
LLM calls	1	5	5
Correct	100%	100%	100%
Recovery tokens	904 ± 17	1,632 ± 145	436 ± 132

Table 2: Multi-file debugging across 10 runs. Recovery cost under simulated Validation failure.

Metric	Mono.	Static	SSM
Tokens (mean ± sd)	703 ± 49	2,181 ± 240	2,225 ± 270
Latency s (mean ± sd)	15.37 ± 18.38	26.28 ± 2.59	21.91 ± 2.70
LLM API (s)	15.37 ± 18.38	21.55 ± 2.13	17.96 ± 2.21
Framework (s)	—	4.73 ± 0.47	3.94 ± 0.49
LLM calls	1	4	4
Correct	100%	100%	100%
Recovery tokens	703 ± 49	933 ± 93	460 ± 77

bounded-context calls replacing one monolithic prompt; the efficiency advantage is realized on recovery paths and per-stage observability, not single-shot throughput.

3.2 Workload 2: Multi-File Code Debugging

The task involves a two-file Python pipeline with three causally ordered bugs: a wrong aggregation operator in `pipeline.py`, an off-by-one sliding window error masked by the first bug, and a type mismatch in `validator.py` unreachable until both are fixed. The SSM pipeline uses four stages; on Validation failure, only Fix Generation is retried over the bug analysis JSON—not the full source files.

Static decomposition increases recovery cost by 32.7% over monolithic (933 vs. 703 tokens), a smaller effect than Workload 1 because the dependency chain is shallower (two cascading stages vs. three). SSM reduces recovery cost to 460 tokens (34.6% below monolithic).

3.3 Design Implications

Recovery cost scales with pipeline depth, not decomposition count. The difference between Workload 1 (+80.5%, 3 cascading stages) and Workload 2 (+32.7%, 2 cascading stages) confirms that static decomposition's recovery penalty is a function of downstream dependency depth. SSM eliminates this scaling relationship by scoping recovery to the failed stage regardless of depth.

Stage boundaries are natural AgentOps instrumentation points. Natural failure data localises the only observed failure to Anomaly Classification (Stage 2) in Workload 1—identifiable without full pipeline replay. In a monolithic configuration, this failure silently degrades the final report with no stage-level attribution.

Per-stage model substitution is structurally available. Framework overhead is consistent across stages (≈18% of wall time), so individual stages can be assigned different models—a smaller model for deterministic format conversion, a larger model for causal inference—without reconfiguring the orchestration layer.

4 HUMAN-IN-THE-LOOP VIA SCHEMA CONTRACTS

SSM's schema contracts create structured human oversight points absent in monolithic pipelines. The state store exposes validated, typed intermediate results at every stage boundary, enabling a human operator to: **inspect** any stage's committed output as a structured record before downstream execution; **override** a misclassified label or wrong root cause attribution and resume from that checkpoint without rerunning upstream stages; or **halt** the pipeline if committed output violates a domain invariant that schema alone cannot express.

When a stage exhausts its retry budget, SSM raises a structured exception containing the stage identifier, last attempted output, and validation error. A human reviewer inspects the bounded context of the failed stage—a small JSON object—rather than a multi-thousand-token prompt, corrects the output, and injects it into the state store to resume execution. This *graceful degradation with human takeover* is qualitatively different from monolithic failure modes, where malformed intermediate outputs silently corrupt the final result.

Each commit generates an audit record (stage ID, schema version, validation outcome, input keys, token count, timestamp)—a *pipeline provenance log* that supports governance requirements in regulated domains such as incident response and financial analysis. A confidence field in the stage output schema can also trigger human review when below a threshold, combining schema-integrity and model-confidence intervention mechanisms into a single checkpoint.

5 RELATED WORK

Agentic pipeline frameworks and coding systems. DSPy [6], LangGraph [8], and AutoGen [20] define pipeline structure statically; branching and recovery are not first-class primitives. DSPy Assertions [18] enforce semantic constraints within a module; our schema contracts enforce integrity *between* stages. SWE-agent [21] and OpenDevin [19] represent the state of the art in agentic software engineering but treat recovery and context management as framework concerns, not first-class data management primitives. Reflexion [17] retries at the full-task level; LLMCompiler [7] parallelises a statically derived task graph. SSM differs by scoping retry to the failed stage and adapting branching at runtime against validated state.

Data management for agentic systems. Recent CIDR work argues that data systems must be fundamentally redesigned for agent-first workloads [13], and that declarative query processing can optimise AI-powered analytics pipelines [11]. SSM addresses the complementary problem: managing the *intermediate state* produced by LLM inference stages, applying schema contracts and dependency-aware recovery to the LLM inference layer of a data

pipeline. Apache Flink [1] achieves fault tolerance through checkpointing; SSM applies analogous principles where recovery targets schema-valid LLM output rather than bit-identical operator results.

Context and memory management. MemGPT [16] addresses the bounded-context problem through virtual memory paging for LLMs; attention degradation over long inputs is well-documented [12]. SSM eliminates unnecessary context re-ingestion structurally via key-based projection, without relying on retrieval or paging at inference time. RAG systems [9] address the same problem at the document level; SSM stages can invoke RAG retrievers as part of their context projection.

Agent reliability and evaluation. MAST [2] taxonomises multi-agent LLM failure modes; our stage-level schema validation directly addresses malformed-output propagation and cascading subtask failures. AgentBench [14], SWE-bench [5], and ITBench [4] evaluate agent capability and IT automation; our evaluation addresses the complementary operational dimension: recovery efficiency, context consumption, and observability.

6 CONCLUSION

We presented SSM, a data-centric pattern that applies typed interfaces, schema-validated persistence, and dependency-aware recovery to agentic pipelines. The key empirical finding: static decomposition can *increase* recovery cost by up to 80.5% over monolithic due to cascading re-reads, while SSM achieves a 73.2% reduction over static and 51.7% over monolithic—maintaining 100% task correctness while providing stage-level audit records for human oversight.

Limitations and future work. Both workloads are at temperature 0 with simulated failures; natural failure rates are low (0–2%). Future work includes evaluation on ITBench [4] and BIRD [10], automatic schema inference from natural-language task descriptions, and learned stage decomposition from execution traces. A Mellea [15]-based reference implementation and workload scripts are available from the authors.

REFERENCES

[1] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin* 38, 4 (2015), 28–38.

[2] Mert Cemri, Melissa Z. Pan, Shuyi Yang, Hamid Palangi, Iñigo Urteaga, Kai-Wei Chang, Cecilia Rosales, et al. 2025. Why Do Multi-Agent LLM Systems Fail?. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[3] FuzzyLabs. 2024. SRE Agent: A Site Reliability Engineer AI Agent. https://github.com/fuzzylabs/sre-agent.

[4] Saurabh Jha, Rohan Arora, Yuji Watanabe, Takumi Yanagawa, Yinfang Chen, Jackson Clark, Mudit Verma, Harshit Kumar, Hirokuni Kitahara, et al. 2025. ITBench: Evaluating AI Agents Across Diverse Real-World IT Automation Tasks. *arXiv preprint arXiv:2502.05352* (2025).

[5] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *arXiv preprint arXiv:2310.06770* (2024).

[6] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, et al. 2024. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *arXiv preprint arXiv:2310.03714* (2024).

[7] Sehoon Kim, Suhong Moon, Rohan Tabrizi, Nicholas Lee, Michael Mahoney, Kurt Keutzer, and Amir Gholami. 2023. An LLM Compiler for Parallel Function Calling. *arXiv preprint arXiv:2312.04511* (2023).

[8] LangChain AI. 2024. LangGraph. https://github.com/langchain-ai/langgraph.

[9] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, et al. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[10] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2023. BIRD: A Big Bench for Large-Scale Database Grounding. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[11] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, et al. 2025. Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing. In *Conference on Innovative Data Systems Research (CIDR)*.

[12] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173.

[13] Shu Liu, Soujanya Ponnapalli, Shreya Shankar, et al. 2026. Supporting Our AI Overlords: Redesigning Data Systems to be Agent-First. In *Conference on Innovative Data Systems Research (CIDR)*.

[14] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hanglei Ding, Kaiwen Men, Kejuan Yang, et al. 2023. AgentBench: Evaluating LLMs as Agents. *arXiv preprint arXiv:2308.03688* (2023).

[15] Mellea Contributors. 2025. Mellea: A Generative Computing Framework for Structured LLM Programs. https://mellea.ai/.

[16] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2023. MemGPT: Towards LLMs as Operating Systems. *arXiv preprint arXiv:2310.08560* (2023).

[17] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[18] Arnav Singhvi, Manish Shetty, Shangyin Tan, Christopher Potts, Koushik Sen, Matei Zaharia, and Omar Khattab. 2024. DSPy Assertions: Computational Constraints for Self-Refining Language Model Pipelines. *arXiv preprint arXiv:2312.13382* (2024).

[19] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, et al. 2024. OpenDevin: An Open Platform for AI Software Developers as Generalist Agents. *arXiv preprint arXiv:2407.16741* (2024).

[20] Qingyun Wu, Gagan Bansal, Jie Zhang, Yiran Wu, Bei Li, Erkang Zhu, et al. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. *arXiv preprint arXiv:2308.08155* (2023).

[21] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*.