Runtime-Structured Task Decomposition for Agentic Coding Systems

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

Bing Zhang*
Zoom
San Jose, California
bing.zhang@zoom.com

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

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

Ruchi Mahindru
IBM Research
Yorktown, New York
rmahindr@us.ibm.com

Abstract

Agentic coding systems increasingly deploy LLMs for multi-step software engineering tasks—debugging, root cause analysis, code review—yet most encode all task logic, control flow, and output generation within monolithic prompts. This leads to brittle behavior, poor debuggability, and expensive full-pipeline reruns when any step fails.

We present *runtime-structured task decomposition*, an architectural pattern in which task partitioning decisions are governed by executable control flow rather than static prompt text, and LLMs are invoked only for narrowly scoped judgment tasks with schema-validated outputs.

We evaluate this pattern on two software engineering workloads across **three configurations**—monolithic, static decomposition (same subtask graph, no runtime branching), and runtime-structured—over 10 runs each. Across both workloads, we find that decomposition structure alone does not reliably reduce retry cost. In the Kubernetes root cause analysis workload, the static baseline's retry cost ($1{,}632 \pm 145$ tokens) exceeds the monolithic baseline (904 ± 17 tokens) by 80.5%, because fixed sequential execution must rerun multiple downstream subtasks. In the multi-file debugging workload, the same effect appears at smaller magnitude (933 vs. 703 tokens). Runtime-structured decomposition, by retrying only the failed subtask, reduces retry cost in both settings—to 436 ± 132 tokens (RCA) and 460 tokens (debugging)—achieving up to a 51.7% reduction over monolithic and a 73.2% reduction over static decomposition. We discuss implications for agentic coding system design, production debugging, and AgentOps workflows.

CCS Concepts

• **Computing methodologies** → **Artificial intelligence**; • **Computer systems organization** → *Architectures*.

*Work done while at IBM Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ACM CAIS 2026, May 26–29, 2026, San Jose, CA
© 2026 Copyright held by the owner/author(s).

Keywords

Task Decomposition, Agentic Coding, LLM Agents, Schema Validation, Programmatic Orchestration, AgentOps, Agentic Software Engineering

ACM Reference Format:
Shubhi Asthana, Bing Zhang, Chad DeLuca, Hima Patel, and Ruchi Mahindru. 2026. Runtime-Structured Task Decomposition for Agentic Coding Systems. In *Proceedings of ACM Workshop on Agentic Software Engineering (ACM CAIS 2026) (ACM CAIS 2026)*. ACM, New York, NY, USA, 5 pages.

1 Introduction

Agentic coding systems—such as those built on AutoGen, LangGraph, and Claude Code—are among the earliest LLM agents deployed in real software engineering workflows. When a debugging agent fails, does it retry the full analysis or just the broken step? When a root cause analysis pipeline produces a malformed output, can it recover without re-ingesting thousands of log lines? These are production engineering questions with architectural answers.

Most existing agentic coding systems encode task logic, control flow, and output generation within monolithic prompts, leading to brittle behavior, limited debuggability, and expensive full-pipeline reruns when failures occur.

Recent frameworks such as DSPy [4], LangGraph [6], and AutoGen [11] have introduced programmatic abstractions for orchestrating LLM calls. However, task decomposition strategies in these frameworks are typically specified statically by the developer—the structure of the task must be known and fixed at design time. When intermediate outputs are invalid or branching conditions emerge at runtime, a static decomposition cannot adapt.

In this paper, we present **runtime-structured task decomposition (RSTD)**, an architectural pattern in which task partitioning decisions are governed by executable control flow and validated intermediate signals rather than static prompt text. Under this paradigm:

(1) LLMs are invoked only for narrowly scoped judgment tasks with schema-validated outputs.
(2) Orchestration, branching, and state management are handled deterministically in code.
(3) Failures are isolated to individual subtasks, enabling selective retry without full pipeline rerun.

We implement this pattern using the Mellea generative computing framework [8] and evaluate it across **three configurations**—monolithic, static decomposition (same subtask structure, no runtime branching), and runtime-structured—over 10 runs each. The three-way comparison isolates the contribution of runtime branching from decomposition structure alone.

A key empirical finding: static decomposition alone does not reliably reduce retry cost relative to monolithic, and can increase it when failures trigger cascading re-execution of downstream subtasks. Its retry cost (1,632 ± 145 tokens) *exceeds* the monolithic baseline (904 ± 17 tokens) by 80.5%, because fixed sequential execution must rerun all downstream subtasks from the point of failure. Runtime-structured decomposition, by retrying only the failed subtask (436 ± 132 tokens), achieves a 51.7% retry cost reduction over monolithic and a 73.2% reduction over static decomposition—demonstrating that runtime branching, rather than decomposition structure alone, is the primary mechanism.

Our contributions are:

(1) A formalization of RSTD as an architectural pattern for agentic coding systems, distinct from both monolithic prompting and static decomposition in DSPy, LangGraph, and AutoGen.
(2) A three-configuration empirical evaluation across 10 runs per setting, with variance quantified, framework overhead separated from LLM API latency, and a static baseline that isolates the contribution of runtime branching.
(3) Operational implications for agentic coding systems: subtask-level monitoring, selective retry under failure, and per-subtask model substitution.

2 Architecture

Runtime-structured task decomposition separates orchestration, state management, and LLM inference into distinct layers. Rather than embedding task structure inside prompts, decomposition decisions are externalized into executable control flow, enabling task structure to be determined dynamically at runtime.

The system consists of three components, illustrated in Figure 1:

- **Decomposition Engine**: Governs runtime task partitioning via developer-authored conditional logic—branching on schema validation outcomes, confidence thresholds, or predecessor completion status. Branching decisions are resolved at runtime against validated intermediate state.
- **Judgment Operators**: Typed LLM calls scoped to a single reasoning task. Each call declares an explicit output schema; validation failure triggers a targeted repair prompt rather than full pipeline rerun.
- **State Manager**: Persists validated intermediate results keyed by subtask identifier. Downstream subtasks access prior state explicitly by key, keeping per-call context bounded.

Branching decisions evaluate three runtime signals: (1) schema validation outcome; (2) output content (non-empty result or confidence above threshold); and (3) subtask completion status. All policies are deterministic conditional logic, making them auditable, unit-testable, and reproducible.

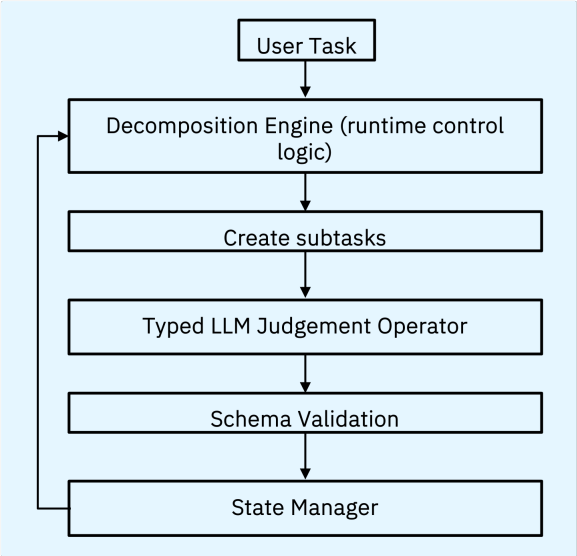

Figure 1: Runtime-structured decomposition architecture. A decomposition engine governs control flow, invoking typed LLM judgment operators over bounded context. Outputs are validated and stored in persistent state, feeding back into subsequent decomposition decisions.

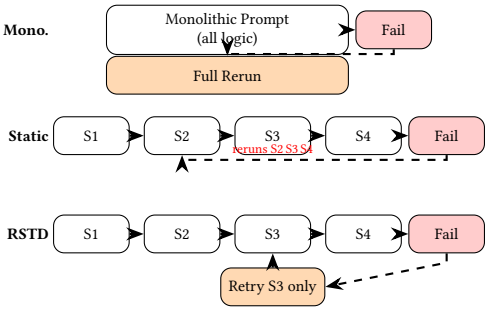

Figure 2: Retry behavior under subtask failure. Monolithic reruns the full pipeline; static reruns all downstream subtasks; RSTD retries only the failed subtask.

We operationalize this architecture using the open-source **Mellea framework** [8], which provides typed LLM calls with enforced input/output schemas, persistent session state, and automated validation-and-repair loops. A Mellea call specifies a prompt and a set of output requirements:

```
result = session.instruct(
  prompt = "Classify anomalies. Output JSON: "
           "[{anomaly, type, severity, conf}]",
  requirements = [
    "Output must be a JSON list.",
    "Each item must have: anomaly, type,
     severity, confidence."
  ]
)
```

```
# On failure, Mellea retries with the
# validation error appended.
```

3 Case Studies

We demonstrate RSTD through two software engineering workloads evaluated across **three configurations**: (i) **Monolithic**—all task logic in a single prompt; (ii) **Static**—same subtask graph, fixed sequential execution, no runtime branching; and (iii) **RSTD**—our approach, with runtime-controlled branching and selective retry. The static baseline isolates whether retry advantages come from decomposition structure alone or specifically from runtime branching.

All agents use `gpt-4` at temperature 0. Token counts are measured using `tiktoken`. Each configuration is run **10 times**; we report mean ± standard deviation. Non-zero standard deviations at temperature 0 reflect API-side variability (e.g., request scheduling and batch-size variation) rather than sampling randomness. Monolithic baselines are drawn from https://github.com/microsoft/autogen and https://github.com/fuzzylabs/sre-agent; structured agent code uses https://mellea.ai/.

3.1 Schema Validation Failure Rate

Natural schema validation failure rates were low: 0% across 40 subtask executions in UC1, and 2.0% (2/100) in UC2, both at Subtask 2 (Anomaly Classification) due to a missing `confidence` field. Retry cost is therefore measured under a *simulated* failure—a structurally malformed upstream input (one missing JSON field)—which measures recovery cost as an architectural property rather than a frequency estimate.

3.2 Use Case 1: Multi-File Code Debugging

The **monolithic** baseline uses AutoGen's `AssistantAgent` [11], encoding bug identification, fix generation, validation, and report synthesis within a single system message.

The task involves a two-file Python pipeline with three causally ordered bugs: (1) a wrong aggregation operator in `pipeline.py`; (2) an off-by-one sliding window error masked by bug 1; (3) a type mismatch in `validator.py` unreachable until bugs 1 and 2 are fixed.

The **static** baseline uses the same four subtasks—Analysis, Fix Generation, Validation, Synthesis—in a fixed sequence. On Validation failure, it reruns Fix Generation and Validation, introducing limited cascading re-execution.

The **runtime-structured** agent uses the same four subtasks but retries only Fix Generation on Validation failure, consuming only the bug analysis JSON—not the full source files.

All three configurations correctly identified and fixed all three bugs across all 10 runs (100% correctness). Results are in Table 1. Static decomposition increases retry cost relative to monolithic (933 vs. 703 tokens, +32.7%), though more modestly than in Use Case 2 due to the shallower dependency chain (two downstream subtasks vs. three). RSTD avoids cascading re-execution by retrying only the failed subtask, reducing retry cost to 460 tokens (34.6% below monolithic).

Note that RSTD's higher baseline token cost (2,225 vs. 703 tokens) reflects the overhead of making multiple API calls for normal successful execution. This overhead is worthwhile when failure rates are non-trivial or when subtasks involve expensive context re-ingestion—the primary deployment scenario this pattern targets.

Table 1: Use Case 1: Multi-file code debugging across 10 runs. Retry cost measured under simulated Validation failure (Section 3.1).

Metric	Mono.	Static	RSTD
Tokens (mean ± sd)	703 ± 49	2181 ± 240	2225 ± 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 (all runs)	100%	100%	100%
Retry tokens	703 ± 49	933 ± 93	460 ± 77

3.3 Use Case 2: Kubernetes Root Cause Analysis

The **monolithic** baseline replicates `fuzzylabs/sre-agent` [1], encoding log parsing, anomaly detection, root cause inference, remediation planning, and report generation within a single prompt.

The task involves a Kubernetes OOMKill on `payment-svc` following deployment v2.4.2, with database connection pool exhaustion and a p99 latency SLA breach. Ground truth: a deployment regression introducing unbounded in-memory batch processing of 84,000 invoice records.

The **static** baseline uses the same five subtasks in a fixed sequence. On Subtask 3 (RCA) failure, it reruns Subtasks 3, 4, and 5.

The **runtime-structured** agent decomposes into five Mellea subtasks, each receiving only the context it requires (Figure 3). On RCA failure, only Subtask 3 is retried.

All configurations correctly identified the root cause across all 10 runs. Results are in Table 2.

Table 2: Use Case 2: Kubernetes root cause analysis across 10 runs. Retry cost measured under simulated RCA failure at Subtask 3 (Section 3.1).

Metric	Mono.	Static	RSTD
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 (all runs)	100%	100%	100%
Retry tokens (mean ± sd)	904 ± 17	1,632 ± 145	436 ± 132

The key finding from Table 2 is counterintuitive: the static baseline's retry cost (1,632 ± 145 tokens) *exceeds* the monolithic baseline (904 ± 17 tokens) by 80.5%, because fixed sequential execution must rerun Subtasks 3, 4, and 5 on failure. RSTD, by retrying only Subtask 3 (436 ± 132 tokens), achieves a **51.7%** retry cost reduction

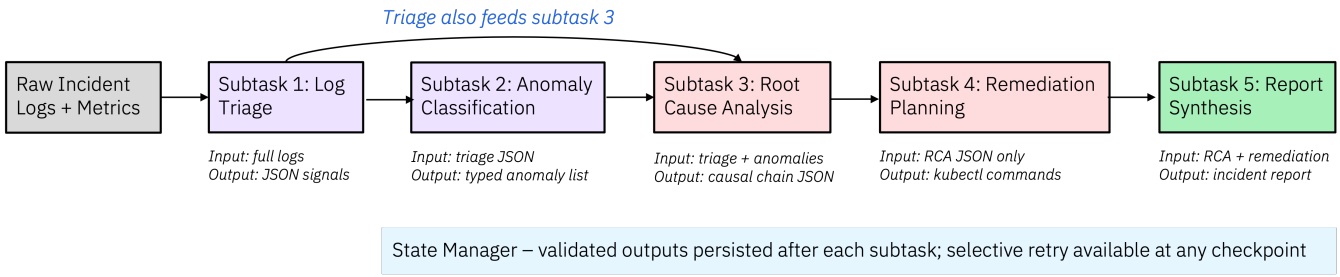

Figure 3: Runtime-structured SRE pipeline. Each subtask receives only predecessor outputs. Triage feeds both Subtask 2 and Subtask 3 (skip arc). Outputs are schema-validated before State Manager writes.

over monolithic and a **73.2%** reduction over static decomposition—confirming that *runtime branching*, not decomposition structure alone, is the primary recovery cost mechanism.

Framework overhead accounts for 4.10–5.18s (∼18% of total latency); the latency gap between monolithic (10.41s) and decomposed agents (∼23–29s) reflects additional API round-trips, not orchestration inefficiency.

3.4 Design Implications

Selective retry reduces recovery cost under failure. RSTD achieves a 73.2% retry cost reduction over static and 51.7% over monolithic. Static decomposition without runtime branching can *increase* retry cost relative to monolithic when pipeline depth causes cascading reruns—in UC2, by 80.5%. The higher RSTD baseline cost is a fixed overhead; whether it is offset by retry savings depends on deployment failure rates.

Subtask-level monitoring enables targeted AgentOps. Each subtask boundary is a natural instrumentation point; natural failure data (Section 3.1) identifies Subtask 2 as the sole natural failure point across both use cases.

Per-subtask model substitution is structurally available. Framework overhead is consistent (∼18% of wall time), so subtasks can be assigned different models without reconfiguring the orchestration layer.

4 Related Work

Agentic coding systems. SWE-bench [3] and AgentBench [7] evaluate agent capability on repository-level tasks. Our work addresses the complementary *operational* gap: not whether agents can solve coding tasks, but how to architect them for reliable, debuggable, cost-efficient operation.

Programmatic LLM frameworks. DSPy [4] proposes declarative LLM programs with typed modules and prompt optimization. LangGraph [6] provides graph-based orchestration for multi-step pipelines. AutoGen [11] enables multi-agent conversation patterns. These frameworks support static decomposition—the task graph is fixed at design time. Our pattern differs in that branching and recovery decisions are resolved at runtime against validated intermediate state, and validation failure blocks downstream state transitions entirely rather than passing malformed outputs forward.

The closest related mechanism is DSPy Assertions [10], which enforce semantic constraints within modules. The key distinction: DSPy Assertions operate within a module and trigger prompt optimization as the primary recovery path. Our validation-gated state transitions operate *between* subtasks—a failed subtask's output is not written to the State Manager and is never visible to downstream subtasks, preventing cascading reasoning errors.

Inference-time control. Reflexion [9] proposes verbal reinforcement for iterative self-correction. Its retry mechanism operates at the level of the full task response. Our validation-gated recovery retries only the failed subtask over its bounded context, providing finer-grained and more token-efficient error correction. LLMCompiler [5] parallelizes a statically derived task graph; our approach adapts branching at runtime without a pre-compiled execution plan.

Production operations for LLM systems. ITBench [2] evaluates agents on IT automation tasks including incident response. Our SRE use case addresses the same domain; future evaluation on ITBench will quantify whether structured decomposition improves accuracy on diverse incident types at scale.

5 Conclusion

We presented RSTD as an architectural pattern that externalizes task structure into executable control flow, enabling selective retry at subtask granularity. Key findings across three configurations over 10 runs: (1) static decomposition does not reliably reduce retry cost—its cost (1,632 ± 145 tokens) exceeds monolithic (904 ± 17 tokens) by 80.5% due to cascading re-execution; (2) RSTD achieves a **51.7%** retry cost reduction over monolithic and **73.2%** over static (436 ± 132 tokens); and (3) framework overhead accounts for ∼18% of total latency.

Limitations. Both use cases are controlled scenarios at temperature 0 with low natural failure rates (0–2%), so retry cost is measured under simulated failure. The higher RSTD baseline cost means overall token savings depend on deployment failure rates, which we do not measure at scale. Decomposition policies are developer-authored and may not generalize to automatically derived graphs.

Future work includes evaluation on ITBench [2] and broader benchmarks, extending to multi-agent settings, and learning decomposition policies from runtime signals.

References

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

[2] Saurabh Jha, Rohan Arora, Yuji Watanabe, Takumi Yanagawa, Yinfang Chen, Jackson Clark, Bhavya Bhavya, 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).
[3] Carlos E. Jimenez, John Yang, Alexander Wettig, et al. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *arXiv preprint arXiv:2310.06770* (2024).
[4] Omar Khattab et al. 2024. DSPy: Programming Language Models Instead of Prompting Them. *arXiv preprint arXiv:2310.03714* (2024).
[5] 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).
[6] LangChain AI. 2024. LangGraph. https://github.com/langchain-ai/langgraph.
[7] Xiao Liu et al. 2023. AgentBench: Evaluating LLMs as Agents. *arXiv preprint arXiv:2308.03688* (2023).
[8] Mellea Contributors. 2025. Mellea: A Generative Computing Framework for Structured LLM Programs. https://mellea.ai/.
[9] 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*.
[10] 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:2312.13382 [cs.CL]
[11] Qingyun Wu, Gagan Bansal, Jie Zhang, Yiran Wu, Bei Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Zhiyuan Liu, et al. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. *arXiv preprint arXiv:2308.08155* (2023).