

Research Article

Dynamic Production Issue Triage Through Declarative Workflows

Kumar Chandan^{1*}

¹422 Deerhill Dr, San Ramon, CA -94583, USA.

*Corresponding author: reachkumarchandan@gmail.com

Article Info

Keywords: Declarative workflows, Intent recognition, LangGraph, Large language models, Orchestration, Production triage, State graphs, Workflow automation.

Received: 03.10.2026;

Accepted: 22.10.2026;

Published: 28.10.2026

 © 2026 by the author's. The terms and conditions of the Creative Commons Attribution (CC BY) license apply to this open access article.

Abstract

Production support teams spend a disproportionate amount of time on repetitive, cross-service triage: correlating order state, provisioning records, logs, and ticket history to answer why a given transaction is broken. Existing runbook automation is typically hard-coded, owned by a single engineering team, and brittle to new failure modes. We present the Workflow Intent Graph System, an architecture that lets any team declare a triage or operational workflow in YAML, compiles that declarative definition into a LangGraph state graph, and uses a large language model (LLM) to match natural-language queries directly to the graph node representing the correct entry point, then navigates the graph at run time to reach a diagnosis or an automated fix. The system supports sub-workflow reuse as edges between independently authored subgraphs, parallel step execution, and dynamic multi-workflow chaining, without requiring workflow authors to write orchestration code. We describe the architecture, the YAML-to-LangGraph compilation step, the node-level intent-resolution pipeline, and the resulting extensibility, auditability, and performance characteristics.

1. Introduction

Modern production systems are decomposed into many independently owned services. When something goes wrong, diagnosing it usually means pulling together data from several of those services in a specific order: check order or transaction state, cross-reference logs, look for related tickets, and inspect downstream fulfillment or provisioning status. Teams typically solve this in one of two ways: engineers write ad hoc scripts or one-off runbooks, or non-engineering operators follow manual, documentation-based procedures. Both approaches share three problems.

Engineering bottleneck: Adding a new triage path requires code changes and a deploy, even when the underlying logic is just calling APIs in a given order and branching on the result.

Poor discoverability: Operators need to already know which script or runbook applies to their symptom.

No composition: Related workflows (e.g., checking acquisition status and checking a reference ID) are usually implemented as separate, non-interoperable tools, so an operator must run several tools by hand and stitch the results together manually.

This paper describes a system designed to remove the engineering bottleneck by making workflows declarative, and to remove the discoverability and composition problems by putting an LLM in front of workflow selection and letting workflows call each other as sub-graphs.

Problem statement: automate triage and multi-step, cross-service processes through natural language, while enabling any team to create new workflows without engineering dependencies.

The contributions of this paper are as follows.

- A declarative (YAML) workflow definition format that is version-controlled like code but requires no orchestration code to execute.
- A compiler that translates workflow definitions into a LangGraph state graph, so intent matching and execution operate over graph primitives rather than an opaque script.
- An LLM-based intent-resolution pipeline that maps a free-text query to a specific graph node and extracts the parameters needed to run from that node.
- A workflow executor that supports parallel step execution, conditional branching, and sub-workflow invocation as graph edges, enabling dynamic graph composition at run time.

2. Background and Motivation

Runbook automation and orchestration are well-established problems, typically solved with workflow engines—state-machine or DAG-based orchestrators [1–3] that execute a workflow once its identity and inputs are already known. That still leaves two open problems in a triage setting: 1) how does the system decide which workflow applies to a given, unstructured symptom report, and 2) how does an operator or another workflow discover and invoke a related workflow without hard-wiring that dependency into the orchestration code?

Our motivation is that both problems can be addressed at the metadata layer rather than the execution layer. If every workflow declares, alongside its steps, a set of natural-language triggers, example queries, and required inputs, an LLM can match a query against that metadata much like a search engine matches a query against a document index. And if a workflow can declare a call to workflow X with given inputs as an ordinary step type, workflow composition becomes a graph-construction problem rather than a code-integration problem.

3. System Architecture

The system is organized into four layers Figure 1 a user interface layer, an LLM intent-detection layer, a workflow executor built on LangGraph, and a tool integration layer that adapts backend services—order and provisioning APIs, log and metrics search, ticketing, fulfillment, and account or subscription systems—into callable activities.

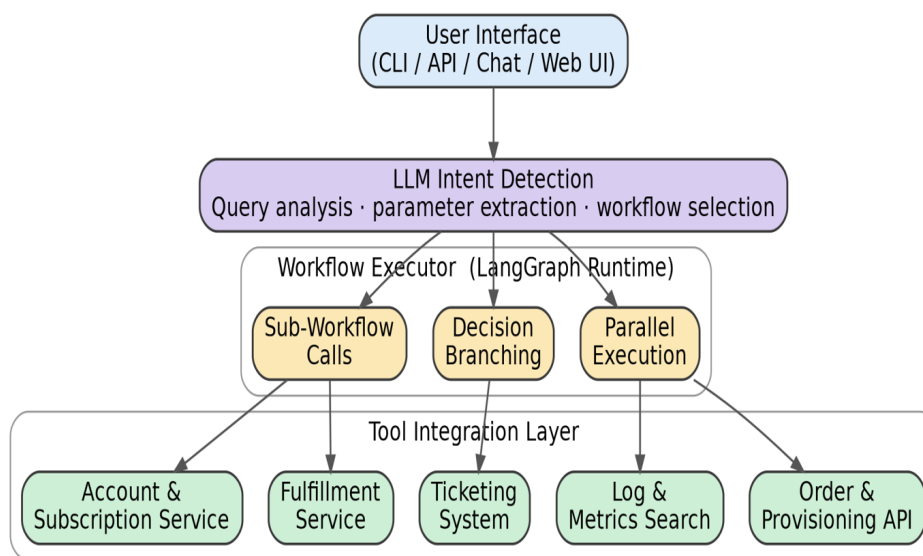


Figure 1: Layered system architecture: user interface, LLM intent detection, the LangGraph-based workflow executor, and the tool integration layer

3.1. User Interface

Queries can originate from a CLI, an internal API, a chat surface, or a web UI. The interface layer is a thin transport; all reasoning happens downstream.

3.2. LLM Intent Detection

This layer analyzes the incoming query, extracts candidate parameters (such as an order or reference ID), and matches the query to the graph node or nodes that best represent the situation described, using the metadata declared by each workflow and node.

3.3. Workflow Executor

Given a matched node and its resolved inputs, the executor navigates the compiled LangGraph beginning at that node. The executor supports three step types beyond ordinary sequential steps: parallel execution, decision branching, and sub-workflow calls (Section 4).

3.4. Tool Integration Layer

Each workflow step ultimately resolves to a call against an integrated backend: an order/provisioning API, a log or metrics search platform, a ticketing system, a fulfillment service, or an account/subscription service. New backends are added by registering a new activity, not by modifying the executor.

3.5. End-to-End Data Flow

Figure 2 shows the full request lifecycle, from an unstructured query to a formatted response: query intake, LLM analysis, workflow selection, graph compilation, execution—with parallel steps, branches, and sub-workflow calls as needed—tool invocation, result aggregation, and response formatting.

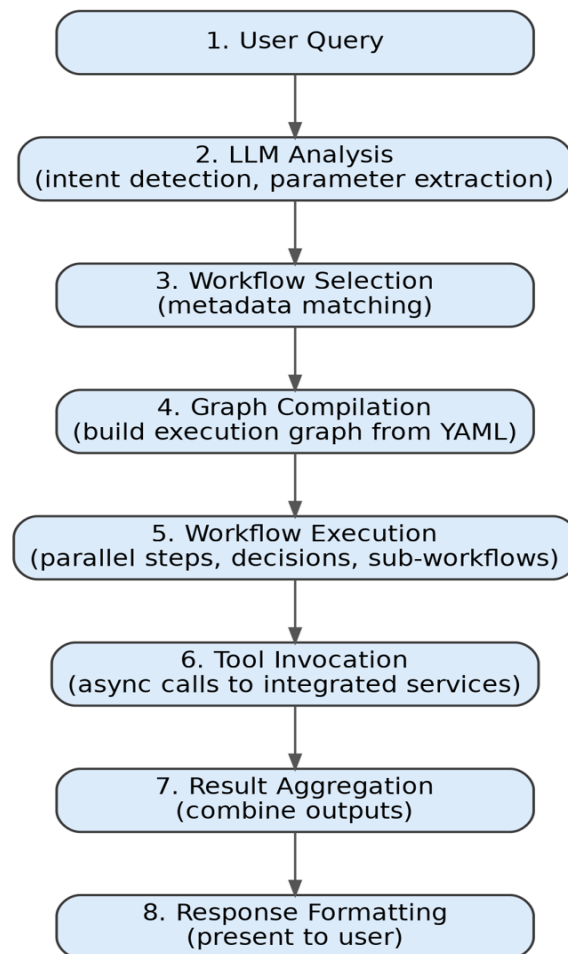


Figure 2: End-to-end data flow from query intake to response formatting

4. Declarative Workflow Definitions and Graph Compilation

4.1. YAML Workflow Definitions

Workflows are authored as YAML documents that are version-controlled alongside application code but interpreted at run time rather than compiled into it. A workflow definition declares metadata used for intent matching (triggers, keywords, and example queries), its required inputs, and an ordered list of steps. Steps can be plain activity calls, parallel groups, conditional branches, or calls to other workflows. Below is a step from an acquisition-pipeline triage workflow that delegates part of its work to a separate reference-ID triage workflow.

```

# acquisition-pipeline-triage.yaml
steps:
  - id: call_reference_triage
    type: workflow
    workflow: reference-id-triage
    inputs:
      reference_id: ${workflow.inputs.
                        reference_id}
  
```

Because a sub-workflow call is just another step type, any workflow can be reused as a component of a larger one without special-casing it in the executor.

4.2. Compiling Workflows into a LangGraph State Graph

YAML is the authoring format, not the execution format. At load time, a compiler translates each workflow’s step list into a LangGraph StateGraph [4] rather than handing it to a bespoke interpreter. Every YAML step type maps onto a small set of LangGraph primitives Figure 3 a plain activity step compiles to a ToolNode that wraps the corresponding backend call; a parallel step compiles to a fan-out to several nodes, joined by an aggregation node before the graph continues; a decision/branch step compiles to a conditional edge, whose router function inspects the graph’s shared state and returns the id of the next node; and a workflow (sub-workflow) step compiles to an edge into the entry node of another team’s already-compiled subgraph, rather than an out-of-band function call.

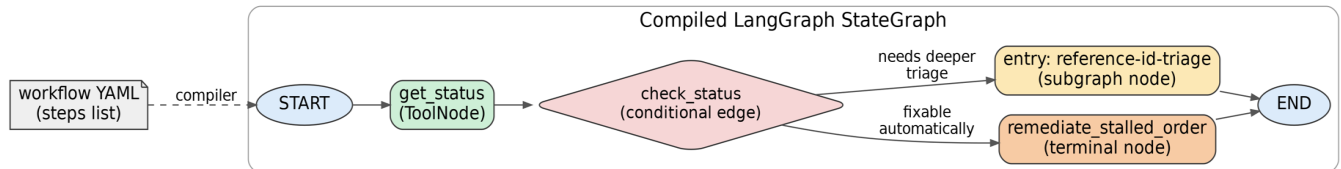


Figure 3: A workflow’s YAML steps compile into LangGraph primitives: tool nodes, a conditional edge, and an edge into a sub-workflow’s entry node.

A shared state object—carrying the original query, extracted parameters, and outputs accumulated from earlier tool nodes—flows along every edge in the graph. This is what makes the parameter-extraction and result-aggregation stages of the intent-resolution pipeline (Section 5) simple reads and writes against shared state, rather than a bespoke data-passing mechanism between steps. Because every workflow compiles into the same primitives, the full corpus of workflows registered by every team is not a set of independent scripts; it is a single, large, addressable graph, with sub-workflow edges being exactly what stitches separately authored subgraphs into one navigable structure.

4.3. Node-Level Intent Matching and Graph Navigation

Because the compiled artifact is a graph rather than an opaque script, trigger and keyword metadata can be attached at the node level, not only at the workflow level. This changes what the LLM is doing during intent resolution: it is not merely choosing a workflow file, it is resolving a query to the specific node in the unified graph that represents the correct entry point for the situation described. A first-time triage question typically resolves to a workflow’s first node; a follow-up query that already implies an earlier step’s outcome can resolve directly to a downstream node, skipping steps whose preconditions are already known to be satisfied.

Once a node is resolved, execution proceeds as graph navigation rather than sequential script execution: the runtime walks outgoing edges from the matched node, following conditional edges based on tool outputs written into shared state and crossing into and out of subgraphs on sub-workflow edges, until it reaches a terminal node. Terminal nodes are not limited to diagnostic reports—a subset are remediation nodes that take a corrective action directly, such as retrying a stalled provisioning call or requeuing a failed fulfillment event. Navigating the graph can therefore end in either a diagnosis or an automated fix, depending on which terminal node the traversal reaches.

5. The Intent-Resolution Pipeline

Query-to-execution proceeds in eight stages Figure 4 the query is passed to the LLM [5], the response is parsed, workflow and node metadata are loaded, keywords are matched against that metadata, candidate nodes are scored and ranked, parameters are extracted from the query, the best-matching node is selected, and the compiled LangGraph is navigated from that node.

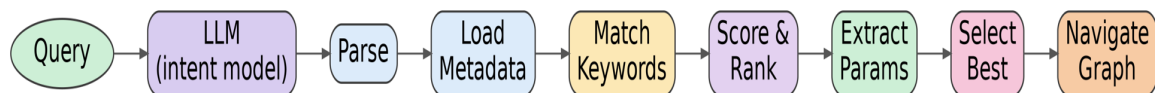


Figure 4: The eight-stage intent-resolution pipeline. The final stage navigates the compiled LangGraph from the matched node rather than re-running a script from the top

This pipeline is deliberately layered rather than a single opaque LLM call: keyword matching and scoring give the system an auditable, debuggable middle stage between free-text input and graph navigation, and let teams tune workflow and node metadata to improve match quality without touching the LLM prompt itself.

6. Key Capabilities

6.1. Workflow and Node Reusability

Because sub-workflow calls are ordinary steps, complex diagnostic procedures can be built from smaller, independently maintained workflows. This reduces duplication—common sub-procedures, such as looking up a reference ID, are written once—and keeps behavior consistent across the workflows that depend on them.

6.2. Interconnected Workflow Discovery

Workflows can chain to other workflows dynamically, based on conditions observed at run time rather than fixed in advance. Because sub-workflow calls compile to edges into another subgraph’s entry node (Section 4.2), this chaining is ordinary graph traversal rather than a special invocation mechanism. Figure 5 illustrates this for the query “Why is my order stuck—AB12345?”: the system matches an acquisition-pipeline-triage node, which checks status and, finding no fulfillment event, crosses the subgraph edge into reference-id-triage rather than terminating with an inconclusive result.

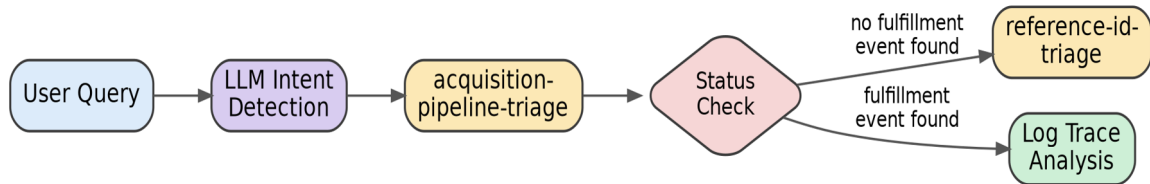


Figure 5: Interconnected workflow discovery: the compiled graph crosses a subgraph edge into a second workflow based on a run-time status check

Example execution:

```

User: "Why is my order stuck - AB12345?"
-> LLM matches node:
    acquisition-pipeline-triage.check_status
-> Graph navigation finds:
    no fulfillment event
-> Crosses subgraph edge into:
    reference-id-triage
-> Reaches terminal node:
    diagnostic result
  
```

6.3. Parallel Execution

Independent activities within a workflow can be declared as a parallel group rather than a sequence, so the executor issues them concurrently.

```

- id: parallel_initial_fetch
  type: parallel
  activities:
    - id: get_provisioning_status
      activity: get_provisioning_
        details_by_reference_id
    - id: analyze_order
      activity: analyze_order_by_
        reference_id
  
```

In our internal usage, workflows that fetch from two or more independent backends see roughly a 2–3x reduction in wall-clock latency compared to the equivalent sequential steps, since the dominant cost is I/O-bound calls to external services rather than local computation.

6.4. Multi-Workflow Composition

A single query can express more than one intent. The LLM can return a ranked list of matching workflows with independent inputs and a priority, rather than being forced to pick exactly one, and the executor can then run these in two modes: parallel, executing all matched workflows simultaneously when they are independent, or sequential, executing them in priority order when one workflow’s output informs the next.

7. Discussion

Extensibility without a shared codebase: Because a new capability is a YAML file plus metadata rather than a code change, teams that own a service can contribute a triage workflow for that service without a dependency on the platform team that owns the executor. This is the central design goal stated in Section 1.

Auditability and guardrails: Letting an LLM select and parameterize workflows raises an obvious question: what stops a misclassified intent from running the wrong workflow, or a malformed extracted parameter from being passed to a production system call? Our pipeline addresses this by keeping the LLM’s role bounded to classification and extraction—it selects among a closed, human-authored set of graph nodes and extracts inputs that are validated against each node’s declared input schema before execution. This matters more once matching can resolve directly to a remediation node with real side effects (Section 4.3): remediation nodes are held to the same authoring and review process as any other node, and low-confidence matches onto remediation nodes are treated as a case for human confirmation before the graph is allowed to cross into them.

Operational trade-offs: The 2–3x latency improvement from parallel execution (Section 4.3) applies specifically to I/O-bound, independent calls; workflows with genuine sequential dependencies do not benefit and should not be forced into a parallel shape. Similarly, multi-workflow composition (Section 6.4) is most useful when a query genuinely spans independent concerns; over-eager multi-intent detection can produce redundant tool calls, so metadata and scoring thresholds need to be tuned per deployment.

8. Related Work

The workflow-executor half of this system is in the same family as general-purpose orchestration engines—state-machine and DAG-based systems used for durable execution and business-process automation [1–3] which we build on for the mechanics of retries, branching, and parallel steps. Concretely, we implement the execution layer on LangGraph [4] workflow authors write YAML, and a compiler emits the StateGraph, node, and conditional-edge definitions that LangGraph executes, so authors get graph semantics without writing graph code by hand. What distinguishes this work from orchestration engines generally is the pairing of that execution model with an LLM-driven intent layer [5] and node-level metadata, so that graph entry-point selection and cross-subgraph composition, not just execution, happens dynamically and without engineering involvement.

9. Limitations and Future Work

The system’s routing quality depends on the quality of workflow and node metadata, which means onboarding a new workflow well requires some care from its author; we are exploring LLM-assisted metadata generation to lower this burden. Node-level metadata is more granular than workflow-level metadata alone, which improves match precision for follow-up queries but increases the authoring surface. We are also evaluating confidence thresholds and human-confirmation steps for cases where the top two candidate nodes score closely, and structured evaluation of intent-resolution accuracy across a held-out set of real triage queries.

10. Conclusion

Treating triage workflows as declarative, LLM-addressable graph fragments—rather than as hard-coded scripts owned by a single team—removes the engineering bottleneck that typically limits how much of production triage can be automated. By combining natural-language intent detection, YAML-based workflow definitions, a LangGraph compilation layer, and a small set of composable step types, the Workflow Intent Graph System lets any team extend triage automation without touching a shared codebase, while keeping execution auditable and bounded to reviewed, version-controlled workflows.

Article Information

Conflict of Interest: The author declare that they have no conflict of interest.

Funding: No external funding was received for this study.

AI Use Disclosure: No generative artificial intelligence tools were used in the preparation of this manuscript.

References

- [1] Temporal Technologies. Temporal documentation, 2024. URL <https://docs.temporal.io/>.
- [2] Argo Project. Argo Workflows documentation, 2024. URL <https://argoproj.github.io/workflows/>.
- [3] Amazon Web Services. AWS Step Functions documentation, 2024. URL <https://docs.aws.amazon.com/step-functions/>.
- [4] LangChain. LangGraph documentation, 2024. URL <https://langchain-ai.github.io/langgraph/>.
- [5] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, et al. GPT-4 technical report. 2023. URL <https://doi.org/10.48550/arXiv.2303.08774>.