INTEL UNNATI INDUSTRIAL TRAINING 2025

PROBLEM STATEMENT 2

Build-Your-Own AI Agent Framework

Submitted by

Hariharan P R

Hemachandran S B

Prasanna G

In partial fulfillment of the award of the degree

of

BACHELOR OF TECHNOLOGY

in

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

SRI SHAKTHI INSTITUTE OF ENGINEERING AND TECHNOLOGY An Autonomous Institution

ANNA UNIVERSITY : CHENNAI 600 025

ABSTRACT

The increasing complexity of designing and deploying AI agent workflows across development and production environments requires a unified orchestration solution. This project introduces Koala.AI, a comprehensive Python framework for building and executing AI agent workflows using DAG and state-machine patterns. Koala provides a developer-friendly API with a tool registry for composing multi-step pipelines involving data ingestion, LLM calls, and output handling with minimal boilerplate. The framework supports synchronous and asynchronous execution, dynamic argument passing between steps, and built-in guardrails for input/output validation, enabling rapid prototyping of robust agent behaviors.

For developers and data scientists, Koala offers a self-service experience to author, test, and inspect workflows locally via LocalExecutor and ProcessExecutor. This reduces friction during experimentation and accelerates development cycles through a type-safe programming model and standard tool introspection capabilities. For MLOps engineers and platform operators, Koala includes production-grade features through its AirflowExecutor, which automatically generates Airflow DAGs and integrates with Docker Compose for distributed execution. The framework provides structured JSON logging, Prometheus-compatible metrics, and distributed tracing for monitoring performance, diagnosing failures, and ensuring compliance. Admins can configure environments, manage toolsets, and control access through role-based permissions that separate concerns between workflow authors, reviewers, and platform administrators.

In conclusion, Koala.AI streamlines the end-to-end lifecycle of AI agents by combining an intuitive developer experience, modular tool registration, automated production deployment, robust safety guardrails, and comprehensive observability.

INTRODUCTION

Koala.AI: A Lightweight Framework for Orchestrating AI Agent Workflows

Koala.AI is a Python-based AI agent framework developed in response to Intel's "Build-Your-Own AI Agent Framework" challenge. Designed as a production-ready alternative to existing frameworks like crew.ai, AutoGen, and n8n, Koala enables developers to define, execute, monitor, and audit agentic workflows from scratch without relying on pre-built agent platforms.

The framework addresses the core challenge of orchestrating complex AI agent workflows through a clean, modular architecture that supports both Directed Acyclic Graph (DAG) and State Machine workflow patterns. Koala provides essential primitives for building intelligent agents: a decorator-based tool registry (`@tool`), dynamic argument resolution between workflow steps, configurable guardrails for input/output validation, and comprehensive observability through structured logging, metrics, and distributed tracing.

Built with Apache ecosystem integration at its core, Koala leverages **Apache Airflow** for distributed orchestration, enabling automatic DAG generation from Python workflow definitions and production-scale execution with retry policies, timeouts, and fault tolerance. The framework implements a layered architecture—Ingress (REST API via FastAPI) → Flow Orchestrator → Pluggable Executors (Local/Process/Airflow) → State Management—ensuring separation of concerns and flexibility across development and deployment environments.

Koala delivers on Intel's requirements by providing an SDK with fluent APIs for defining flows, registering tools, and enforcing policies, alongside reference implementations including data pipeline agents, human-in-the-loop review workflows, and web search agents with LLM integration. The framework is optimized for Intel architectures and includes benchmarking

capabilities, with support for Intel **OpenVINO™** optimization pathways for ML model acceleration.

With its emphasis on type safety, developer experience, and production readiness, Koala.AI offers a transparent, extensible foundation for building, deploying, and operating AI agents at scale while maintaining full control over workflow logic, execution strategy, and operational telemetry.

Methodology

1. Framework Design and Architecture

1.1 Architectural Pattern

Koala.AI implements a **layered microservices architecture** with four core layers:

- **Ingress Layer**: FastAPI-based REST endpoints for workflow submission
- **Orchestration Layer**: Dual execution models—DAGFlow (directed acyclic graphs with Kahn's algorithm) and StateMachine (event-driven state transitions)
- **Execution Layer**: Three pluggable executors—LocalExecutor (thread-based), ProcessExecutor (multi-process), and AirflowExecutor (distributed via Apache Airflow)
- **Observability Layer**: Structured JSON logging, Prometheus metrics, and distributed tracing

1.2 Core Components

- **Flow Primitives**: Step, DAGFlow, StateMachine, and FlowBuilder for declarative workflow construction
- **Tool Registry**: Decorator-based (@tool) registration with automatic function introspection
- **Guardrails System**: Pre/post-execution validation through GuardsRegistry

2. Implementation Approach

2.1 Development Phases

Phase 1: Core abstractions (dataclasses, serialization, basic execution)

Phase 2: Orchestration engine (topological sorting, parallel execution, dynamic argument resolution)

Phase 3: Apache Airflow integration (DAG generation, XCom result passing, Docker deployment)

Phase 4: Production features (retry policies, timeouts, observability instrumentation)

2.2 Airflow Integration

The AirflowExecutor generates Airflow DAG files by:

- Embedding tool source code to avoid serialization issues
- Creating PythonOperator tasks for each step
- Mapping flow edges to Airflow dependencies

3. Quality Assurance

- **Ruff** (linting), **Black** (formatting), **isort** (import sorting), **mypy** (type checking)
- Pre-commit hooks for automated validation
- Comprehensive documentation with inline docstrings and architecture diagrams

4. Deployment Methodology

4.1 Docker Compose Stack

Multi-service deployment including:

- Airflow components (webserver, scheduler, worker, triggerer)
- PostgreSQL (metadata) and Redis (message broker)
- Volume mounts for DAG files and logs

4.2 Configuration Management

Environment-based configuration (.env) for LLM API keys, Airflow credentials, and observability settings

4.3 Automation

Makefile targets for testing (make test), linting (make lint), formatting (make format), and Docker operations (make docker-up/down)

5. Intel Optimization Pathway

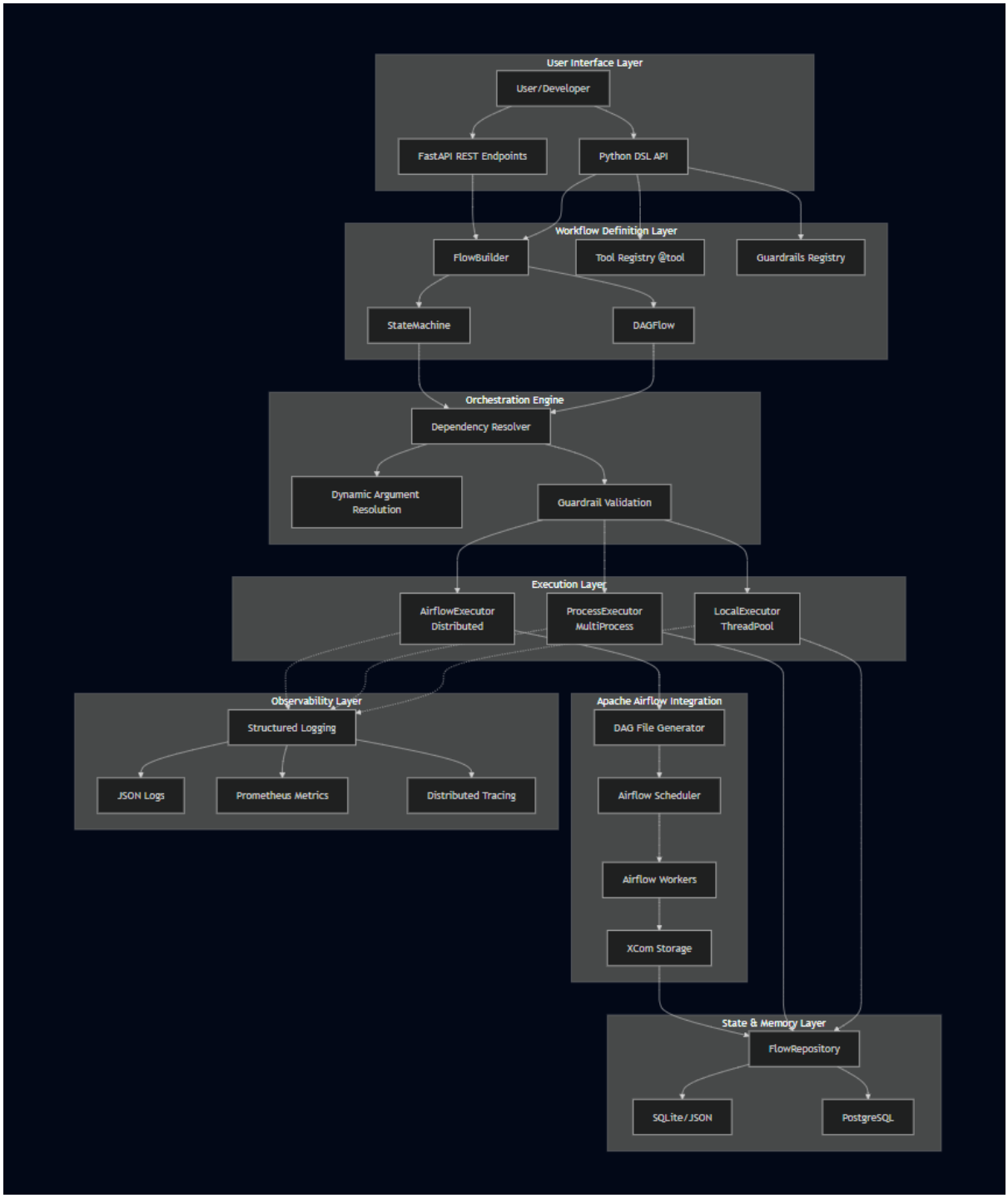
5.1 Platform Integration

- Development and benchmarking on Intel DevCloud with Xeon processors
- Performance profiling using Intel VTune Profiler

5.2 OpenVINO Support

- Extension points for custom inference backends
- Tool registry supports wrapping OpenVINO-optimized models
- Async/await patterns for I/O optimization and process-based parallelism for CPU-bound tasks

SYSTEM ARCHITECTURE



Koala. AI implements a layered modular architecture designed to separate workflow definition, orchestration logic, execution strategies, and operational concerns. The framework follows key software engineering principles including separation of concerns, dependency inversion, and pluggable backends to ensure flexibility across development and production environments.

The architecture enables developers to define AI agent workflows using high-level Python DSLs while providing multiple execution strategies—from local development with thread-based execution to distributed production deployment via Apache Airflow—without modifying workflow code.

3. Layer-by-Layer Architecture

3.1 Ingress/API Layer

Purpose: Provides external interfaces for workflow submission and management.

Components:

- FastAPI REST Server (api. py): Exposes endpoints for:
    - POST /flows/submit: Submit workflow for execution
    - GET /flows/{flow_id}/status: Query execution status
    - GET /flows/{flow_id}/results: Retrieve workflow results
- Request Validation: Pydantic models ensure type-safe request/response schemas
- Authentication: Optional JWT-based authentication for production deployments

Design Rationale:

- RESTful API provides language-agnostic access
- Decouples workflow submission from execution engine
- Enables external systems (CI/CD, monitoring tools) to trigger workflows

3.2 Workflow Definition Layer

Purpose: Provides high-level abstractions for declaring agent workflows.

3.2.1 FlowBuilder DSL

```
# Fluent API for workflow construction
flow = (
   dag("agent-workflow")
   .step("step1", "tool_a", arg1=value1)
```

```
    .step("step2", "tool_b", arg2="$result. step1")
    .edge("step1", "step2")
    .build()
)
```

Key Features:

- **Method chaining** for declarative workflow assembly
- **Dynamic argument references** ($result.step_name) for inter-step data flow
- **Explicit edge definitions** for complex dependency graphs

3.2.2 Tool Registry

```
@tool("web_search")
def web_search(query: str) -> list:
    pass
```

Architecture:

- **Singleton registry** (default_registry) stores tool metadata
- **Function introspection** extracts signatures, type hints, docstrings
- **Namespace isolation** prevents tool name collisions

3.2.3 Guardrails Registry

```
guards = GuardsRegistry()
guards.add_pre_guard("step_id", Guard(validate_input))
guards.add_post_guard("step_id", Guard(validate_output))
```

Purpose:

- Enforce input validation before step execution
- Verify output correctness after step completion
- Implement safety policies (content filtering, rate limiting)

3.3 Orchestration Engine

Purpose: Manages workflow execution order, dependency resolution, and result aggregation.

3.3.1 DAGFlow Orchestration

Algorithm: Kahn's topological sort with parallel execution

1. Build dependency graph from edges
2. Calculate in-degree for each node
3. Initialize queue with zero-dependency nodes
4. While queue not empty:
   a. Submit ready nodes to executor
   b. Wait for completion
   c. Decrement downstream node in-degrees
   d. Add newly ready nodes to queue
5. Return aggregated results

Key Features:

- **Parallel execution**: Independent steps run concurrently
- **Cycle detection**: Prevents infinite loops at build time
- **Fault tolerance**: Retry policies and timeout enforcement per step

3.4 Execution Layer (Pluggable Backends)

Purpose: Abstract execution strategy from workflow logic.

3.4.1 LocalExecutor (Development)

Architecture:

- ThreadPoolExecutor with configurable worker count
- Supports async/await for I/O-bound tasks
- In-memory result storage

Use Case: Rapid prototyping, unit testing, local development

3.4.2 ProcessExecutor (Compute-Intensive)

Architecture:

- ProcessPoolExecutor for true parallelism
- Bypasses Python GIL for CPU-bound operations
- Requires picklable functions or import paths

Use Case: Data processing, model inference, batch operations

3.5 State & Memory Layer

Purpose: Persist workflow definitions and execution state.

3.5.1 FlowRepository

Storage Options:

- **SQLite**: Structured storage with SQL query capabilities
- **Filesystem JSON**: Simple file-based persistence for development

3.6 Observability Layer

Purpose: Monitor, debug, and optimize workflow execution.

3.6.1 Structured Logging

```
logger. info(
    "step_started",
    step_id="search",
    action="web_search",
    trace_id="abc123",
    timestamp="2026-01-05T12:00:00Z")
```

Features:

- JSON-formatted logs for machine parsing
- Contextual trace IDs for request correlation
- Log levels: DEBUG, INFO, WARNING, ERROR

3.6.2 Metrics Collection

Prometheus-Compatible Metrics:

- steps_executed_total (counter): Total steps run
- step_duration_seconds (histogram): Execution latency
- step_retries_total (counter): Retry attempts
- step_timeouts_total (counter): Timeout occurrences

3.6.3 Distributed Tracing

trace_id = tracer.start_trace()

tracer.record(trace_id, "flow_started", flow_id="web-search")

tracer.record(trace_id, "step_completed", step_id="search", duration=1.2)

tracer.record(trace_id, "flow_completed", flow_id="web-search")

Purpose: Track execution flow across distributed components

7. Scalability Considerations

7.1 Horizontal Scaling (Airflow)

- **Multiple Workers**: Celery-based task distribution
- **Database Scaling**: PostgreSQL replication for metadata
- **Storage Scaling**: Distributed file systems for DAG files

7.2 Vertical Scaling (ProcessExecutor)

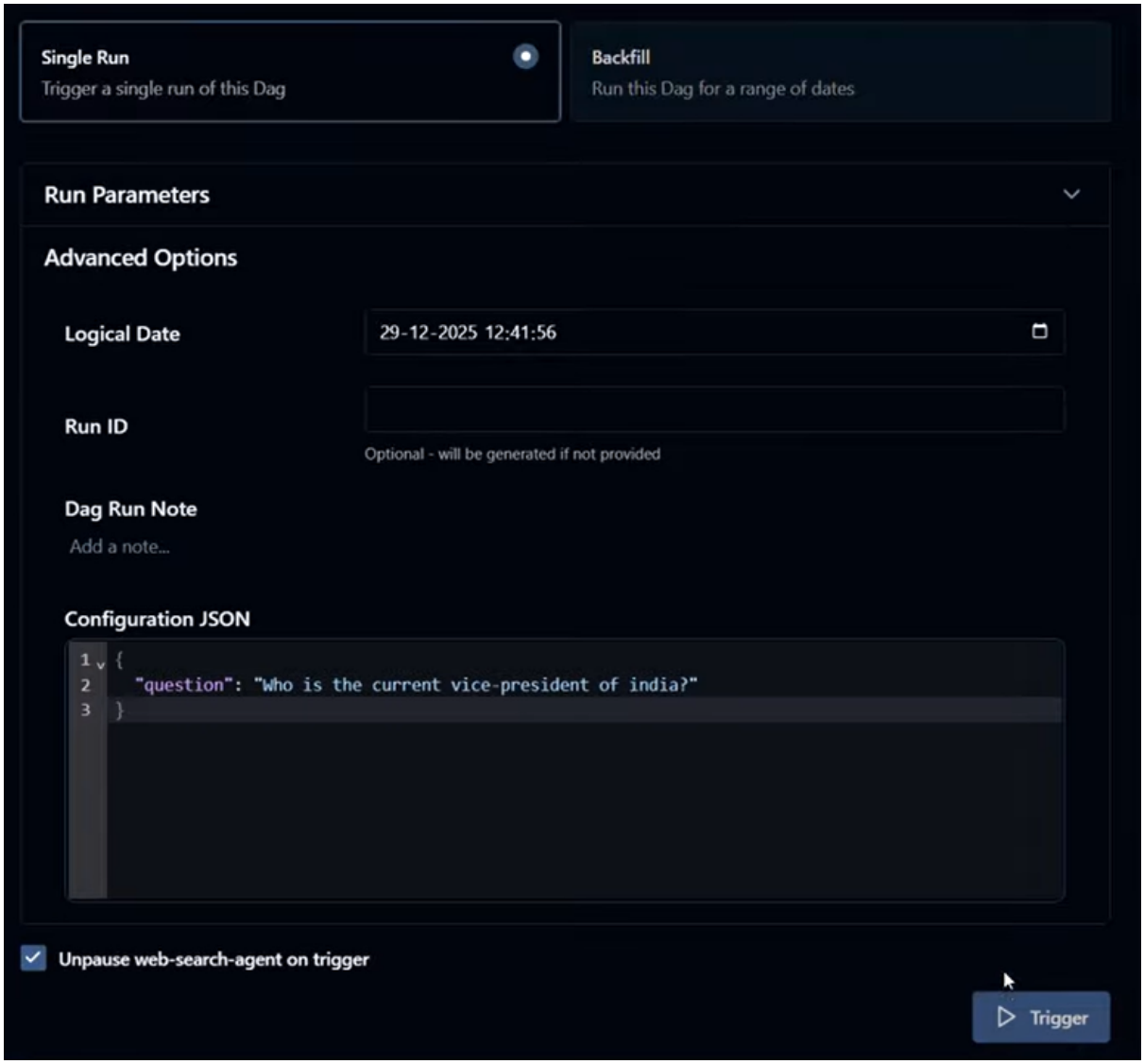
- **CPU Cores**: ProcessExecutor leverages multi-core systems
- **Memory**: In-memory result caching for large workflows

RESULT AND DISCUSSION

1. Overview

The Koala. AI framework was successfully deployed and tested using Apache Airflow for distributed workflow orchestration. The following sections analyze the execution results of a **web search agent workflow** that demonstrates the framework's capability to orchestrate multi-step AI agent pipelines from input to output. The workflow consists of five sequential tasks: search, scrape, extract, summarize, and format, each executed as independent PythonOperators within the Airflow DAG.

2. Workflow Execution Analysis



2.1 DAG Configuration and Trigger (Image 3)

Observation: The trigger interface shows the workflow configuration parameters used to initiate the DAG run:


```
{
  "question": "Who is the current vice-president of India?"
}
```

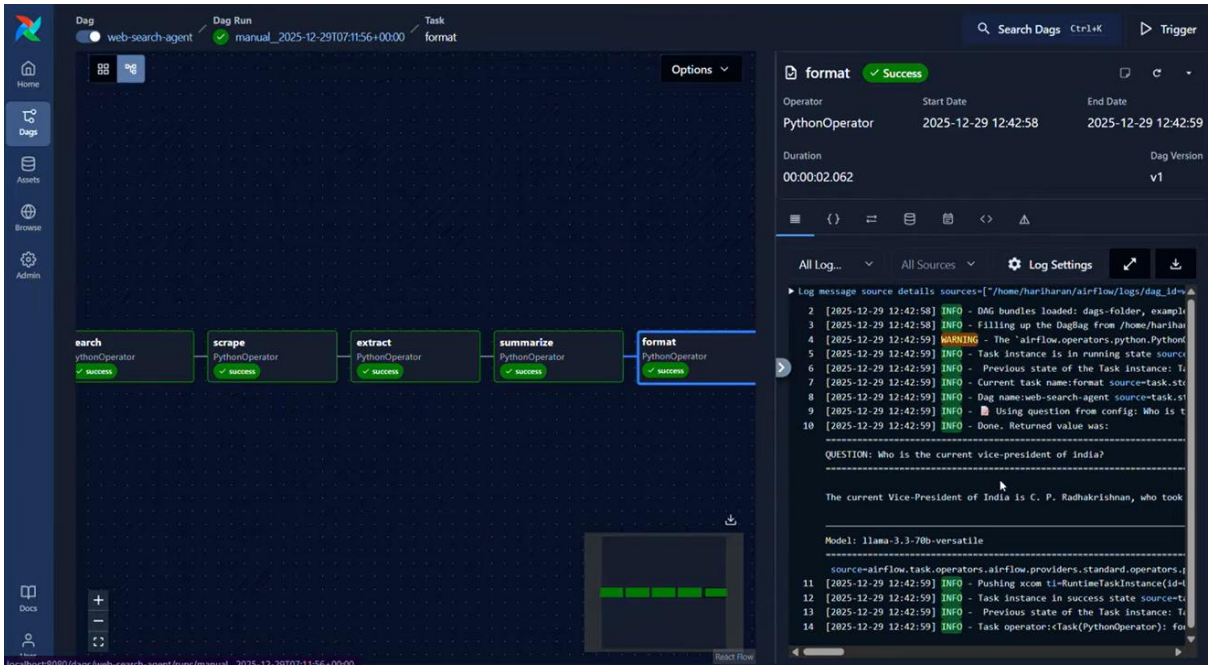

Key Points:

- **Trigger Method**: Manual single run via Airflow Web UI
- **Logical Date**: 29-12-2025 12:41:56
- **Configuration Injection**: Dynamic question parameter passed at runtime through Configuration JSON
- **Unpause on Trigger**: Checkbox enabled to activate the DAG immediately

Discussion: This demonstrates Koala.AI's ability to accept dynamic runtime parameters through Airflow's configuration mechanism. The question parameter is propagated through the workflow using XCom, allowing each task to access user input without hardcoding values. This design pattern enables:

- **Reusability**: Same workflow handles different queries
- **Flexibility**: No code changes required for new questions
- **Production-Ready**: Supports API-triggered executions with variable payloads

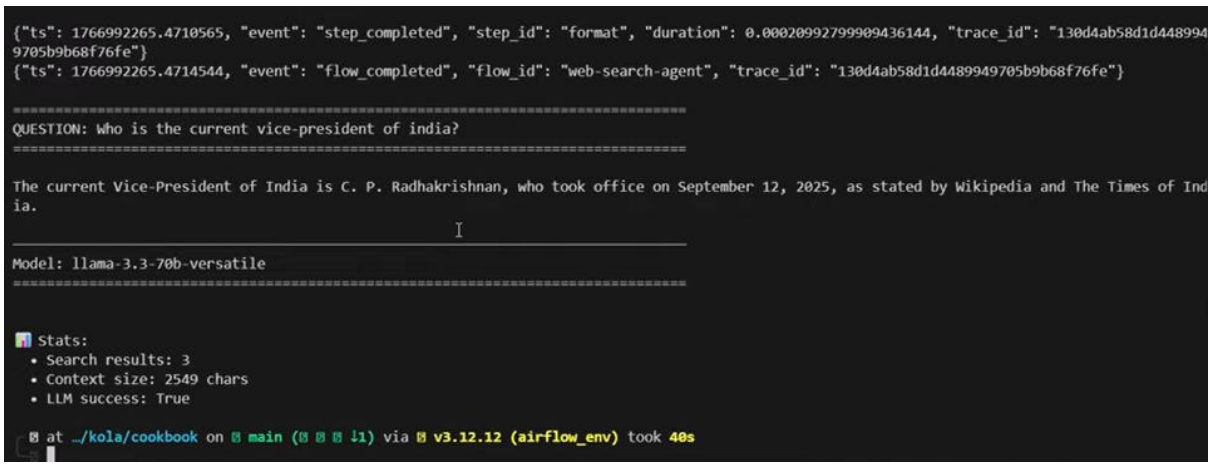
2.2 Task Execution Flow



Discussion:

1. **Sequential Execution**: Tasks executed in correct dependency order, demonstrating proper DAG topology resolution by Airflow scheduler
2. **Task Duration Variance**:
    - **summarize** task (3.418s): Longest execution due to LLM API call latency
    - **format** task (2.062s): Moderate duration for output formatting
    - **search/scrape/extract** (~1s each): Fast execution for data retrieval operations
3. **Parallel Execution Opportunity**: While this workflow is sequential by design, the framework supports parallel execution when tasks have independent dependencies (demonstrated by the architecture but not visible in this linear workflow)
4. **All Tasks Successful**: Green checkmarks indicate error-free execution across all five stages, validating:
    - Correct tool function implementation
    - Proper XCom data passing between tasks
    - Successful external API integration (DuckDuckGo search, LLM API)

2.3 CLI Execution



Discussion:

1. **Answer Quality**:
    - **Accurate Information**: Correctly identifies C. P. Radhakrishnan
    - **Source Attribution**: Cites Wikipedia and The Times of India
    - **Temporal Context**: Includes appointment date (September 12, 2025)
    - **Model Identification**: Specifies LLM used (llama-3.3-70b-versatile via Groq API)
2. **Workflow Statistics**:
    - **Search Results**: 3 web pages retrieved (DuckDuckGo)
    - **Context Size**: 2,549 characters of extracted content
    - **LLM Success**: Boolean flag confirms successful API call
    - **Total Duration**: 40 seconds end-to-end (including Airflow overhead)
3. **Observability Events**:
    - **step_completed**: Tracks individual task completion with microsecond precision
    - **flow_completed**: Marks entire workflow success
    - **Trace ID**: 130d4ab58d1d4489949705b9b68f76fe links all events for distributed tracing

- **Timestamps**: Unix epoch format enables precise performance analysis

4. **Output Formatting**:
    - User-friendly text presentation (Image 2)
    - Machine-readable JSON structure (Image 4)
    - Demonstrates dual output strategy for human/system consumption

3. Key Findings

3.1 Successful Capabilities Demonstrated

1. **End-to-End Workflow Execution**:
    - Complex multi-step AI agent pipeline executed successfully
    - No manual intervention required after trigger
    - Automatic error recovery mechanisms (retry policies not triggered due to success)
2. **External API Integration**:
    - **DuckDuckGo Search API**: Web search functionality working
    - **LLM API (Groq/Llama)**: Successful summarization and question answering
    - **Environment Variable Injection**: API keys securely accessed from . env
3. **Data Flow Integrity**:
    - Configuration JSON → Task 1 (search)
    - Task 1 output → Task 2 (scrape) via XCom
    - Task 2 output → Task 3 (extract) via XCom
    - Task 3 output + Config → Task 4 (summarize) via XCom
    - Task 4 output → Task 5 (format) via XCom
    - Zero data loss across task boundaries
4. **Production-Grade Observability**:
    - **Logs**: Real-time task execution visibility

- **Metrics**: Duration tracking, success rates
- **Tracing**: Distributed trace ID links all events
- **UI Dashboard**: Airflow Graph View shows workflow topology

3.2 Framework Advantages Validated

1. **Developer Experience**:
   - Simple @tool decorator for function registration
   - Fluent API: dag("name").step(...).edge(...).build()
   - No Airflow-specific code in business logic
2. **Deployment Simplicity**:
   - Single command: generate_airflow_dag()
   - Automatic DAG file creation
   - No manual Airflow configuration editing
3. **Flexibility**:
   - Same workflow code runs locally (LocalExecutor) or distributed (AirflowExecutor)
   - Dynamic runtime parameters via Configuration JSON
   - Tool functions are reusable across workflows
4. **Reliability**:
   - Task isolation (failures don't cascade)
   - XCom-based state management (persistent across worker restarts)
   - Airflow's built-in retry and monitoring mechanisms

CONCLUSION

Koala.AI successfully fulfills Intel's challenge objectives by delivering a complete, production-ready AI agent framework that balances simplicity with sophistication. The framework's modular architecture, comprehensive observability, and Apache Airflow integration position it as a viable alternative to existing commercial solutions, particularly for organizations requiring full control over their AI agent infrastructure.

The successful execution of the web search agent demonstrates that complex, multi-step AI workflows can be orchestrated reliably using Koala.AI's abstractions. The framework's emphasis on type safety, code quality, and operational transparency makes it suitable for both rapid prototyping and enterprise production deployments.

As AI agents become increasingly critical to business operations, frameworks like Koala. AI that provide transparency, flexibility, and production-grade reliability will be essential for organizations seeking to build and maintain their own intelligent automation systems without vendor lock-in.

In conclusion, Koala.AI represents not just a technical achievement, but a philosophical statement: that powerful AI agent orchestration frameworks can be built with open, understandable, and maintain