



Workflows & Orchestration

Description: Understanding sequential, parallel, and loop workflow patterns for complex agent orchestration

Workflows & Orchestration

 **Purpose:** Master workflow patterns to orchestrate complex agent behaviors and multi-step processes.

 **Source of Truth:** [google/adk-python/src/google/adk/agents/workflow_agents/](https://github.com/google/adk-python/tree/main/src/google/adk/agents/workflow_agents/)
(https://github.com/google/adk-python/tree/main/src/google/adk/agents/workflow_agents/) (ADK 1.15)

[FLOW] Workflow Patterns Overview

Mental Model: Workflows are like **assembly line strategies** for agent orchestration:

WORKFLOW PATTERNS

[INSTR] SEQUENTIAL (Assembly Line)

"One step after another, in order order"

Step 1 → Step 2 → Step 3 → Step 4
 Write Review Refactor Test

Use: Pipelines, dependencies, order matters

Pattern: Each step uses output from previous

Source: agents/workflow_agents/sequential_agent.py

[PARALLEL] PARALLEL (Fan-out/Gather)

"Multiple tasks at once, then combine"

```

  ┌── Task A ──┐
  ┌── Task B ──┐ → Merge Results
  ┌── Task C ──┐
  Research      Research  Synthesis
  Source 1      Source 2
  
```

Use: Independent tasks, speed critical

Pattern: Fan-out → Execute → Gather

Source: agents/workflow_agents/parallel_agent.py

[LOOP] LOOP (Iterative Refinement)

"Repeat until good enough or max iterations"

```

  ┌────────────────┐
  │ ┌── Critic ──┐ │
  │ │            │ │
  │ └── Refiner ─┘ │
  └────────────────┘
  
```

(Repeat 5x or until exit_loop)

Use: Quality improvement, retry logic

Pattern: Generate → Critique → Improve → Repeat

Source: agents/workflow_agents/loop_agent.py

[INSTR] Sequential Workflows (Assembly Line)

Basic Sequential Pattern

Mental Model: Steps execute in order, each using the output of the previous:

```
from google.adk.agents import SequentialAgent

# Define individual agents
research_agent = Agent(
    name="researcher",
    model="gemini-2.5-flash",
    instruction="Research the given topic thoroughly",
    output_key="research_results"
)

writer_agent = Agent(
    name="writer",
    model="gemini-2.5-flash",
    instruction="Write a comprehensive article based on the research: {research_results}",
    output_key="article_draft"
)

editor_agent = Agent(
    name="editor",
    model="gemini-2.5-flash",
    instruction="Edit and improve the article: {article_draft}",
    output_key="final_article"
)

# Create sequential workflow
content_pipeline = SequentialAgent(
    name="content_creation_pipeline",
    sub_agents=[research_agent, writer_agent, editor_agent],
    description="Complete content creation from research to publication"
)
```

Sequential Workflow Execution

Execution Flow:

User Query → Research Agent → Writer Agent → Editor Agent → Final Result

1. Research agent gets user query
2. Research agent saves results to state['research_results']
3. Writer agent reads {research_results} from instruction
4. Writer agent saves draft to state['article_draft']
5. Editor agent reads {article_draft} from instruction
6. Editor agent produces final output

| Advanced Sequential Patterns

Conditional Branching:

```
# Dynamic routing based on content type
def route_by_topic(context, result):
    topic = result.get('topic', '').lower()
    if 'technical' in topic:
        return 'tech_writer'
    elif 'business' in topic:
        return 'business_writer'
    else:
        return 'general_writer'

routing_agent = Agent(
    name="router",
    model="gemini-2.5-flash",
    instruction="Analyze the topic and determine content type",
    output_key="topic_analysis"
)

tech_writer = Agent(name="tech_writer", ...)
business_writer = Agent(name="business_writer", ...)
general_writer = Agent(name="general_writer", ...)

# Sequential with dynamic agent selection
content_workflow = SequentialAgent(
    sub_agents=[routing_agent], # Start with router
    dynamic_agents={
        'tech_writer': tech_writer,
        'business_writer': business_writer,
        'general_writer': general_writer
    },
    routing_function=route_by_topic
)
```

Parallel Workflows (Fan-out/Gather)

| Basic Parallel Pattern

Mental Model: Independent tasks execute simultaneously, then results are combined:

```
from google.adk.agents import ParallelAgent

# Research different aspects in parallel
web_research_agent = Agent(
    name="web_researcher",
    model="gemini-2.5-flash",
    tools=[google_search],
    instruction="Research topic using web search",
    output_key="web_findings"
)

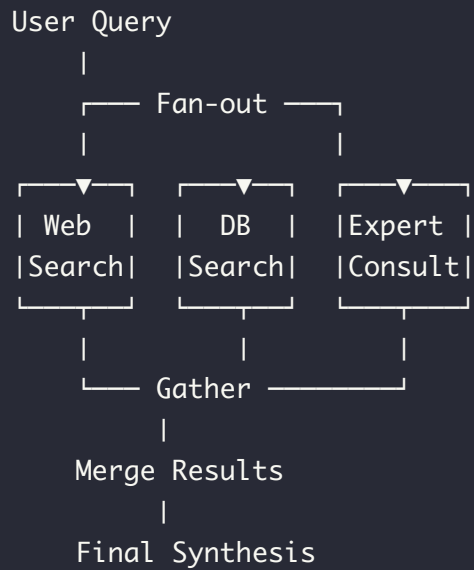
database_research_agent = Agent(
    name="db_researcher",
    model="gemini-2.5-flash",
    tools=[database_tool],
    instruction="Search internal database for relevant data",
    output_key="db_findings"
)

expert_opinion_agent = Agent(
    name="expert_consultant",
    model="gemini-2.5-flash",
    tools=[expert_tool],
    instruction="Consult domain experts on the topic",
    output_key="expert_insights"
)

# Execute all research in parallel
parallel_research = ParallelAgent(
    name="comprehensive_research",
    sub_agents=[web_research_agent, database_research_agent, expert_opinion_ag
    description="Research topic from multiple sources simultaneously"
)
```

Parallel Execution Flow

Fan-out → Execute → Gather:



Parallel with Sequential Merger

Complete Research Pipeline:

```

# Parallel research phase
parallel_research = ParallelAgent(
    sub_agents=[web_agent, db_agent, expert_agent]
)

# Sequential synthesis phase
synthesis_agent = Agent(
    name="synthesizer",
    model="gemini-2.5-flash",
    instruction="""
Synthesize findings from multiple sources:
Web: {web_findings}
Database: {db_findings}
Experts: {expert_insights}

Create a comprehensive report.
""",
    output_key="final_report"
)

# Complete workflow: Parallel → Sequential
research_pipeline = SequentialAgent(
    sub_agents=[parallel_research, synthesis_agent]
)
  
```

Loop Workflows (Iterative Refinement)

| Basic Loop Pattern

Mental Model: Repeat until quality standards are met or max iterations reached:


```
from google.adk.agents import LoopAgent

# Content generator
writer_agent = Agent(
    name="content_writer",
    model="gemini-2.5-flash",
    instruction="Write content on the topic: {topic}",
    output_key="content_draft"
)

# Quality critic
critic_agent = Agent(
    name="content_critic",
    model="gemini-2.5-flash",
    instruction="""
    Evaluate the content quality: {content_draft}

    Rate on scale 1-10 for:
    - Accuracy
    - Completeness
    - Clarity
    - Engagement

    If score < 8, provide specific improvement suggestions.
    """,
    output_key="critique"
)

# Improvement refiner
refiner_agent = Agent(
    name="content_refiner",
    model="gemini-2.5-flash",
    instruction="""
    Improve the content based on critique: {critique}
    Original: {content_draft}

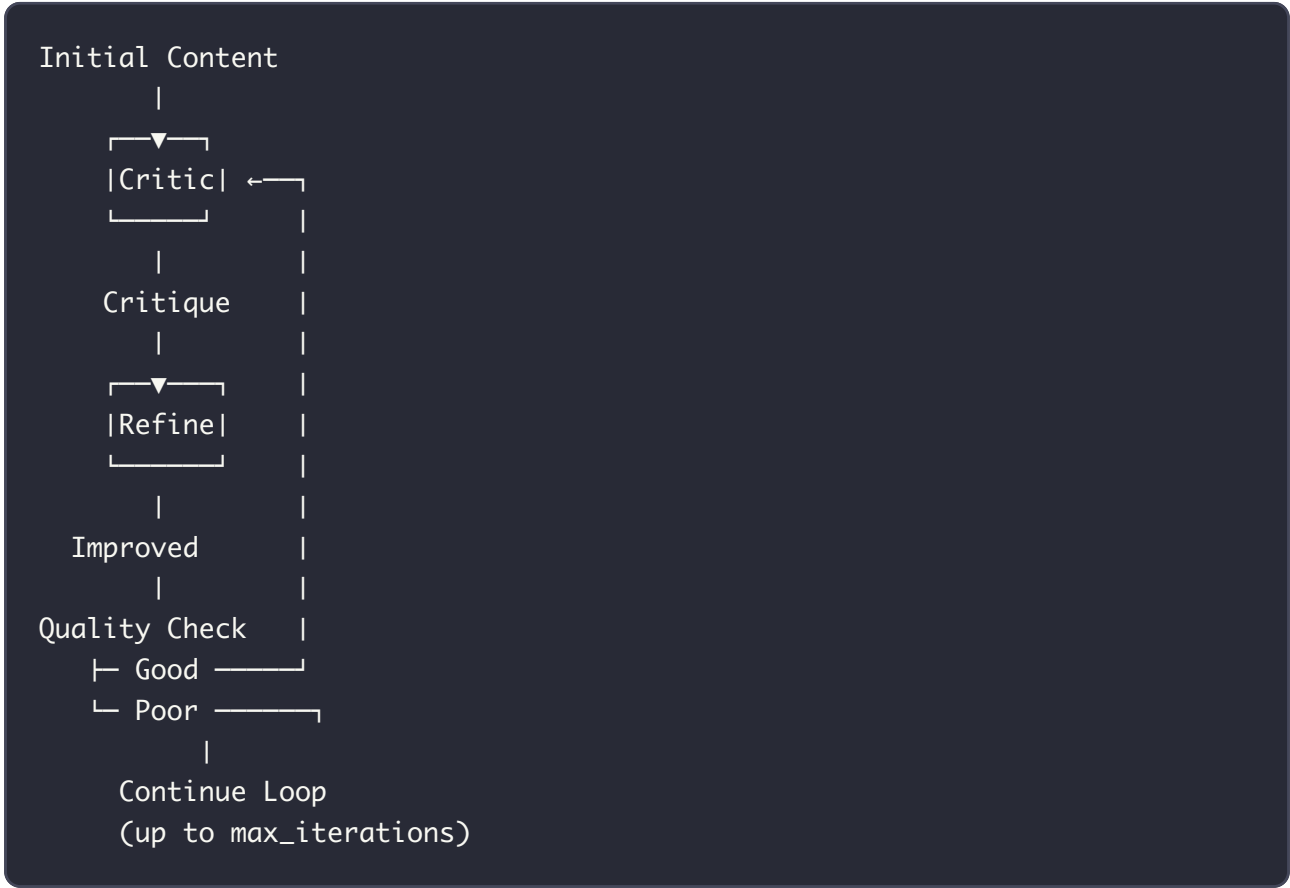
    Address all the critic's suggestions.
    """,
    output_key="improved_content"
)

# Iterative refinement loop
quality_loop = LoopAgent(
    sub_agents=[critic_agent, refiner_agent],
    max_iterations=5,
```

```
description="Iteratively improve content until quality standards are met"  
)
```

| Loop Execution Flow

Generate → Critique → Refine → Repeat:



| Advanced Loop Patterns

Conditional Exit:

```
def should_continue_loop(context, result):  
    """Custom exit condition"""  
    critique = result.get('critique', '')  
    score = extract_score_from_critique(critique)  
    return score < 8 # Continue if quality < 8/10  
  
quality_loop = LoopAgent(  
    sub_agents=[critic_agent, refiner_agent],  
    max_iterations=5,  
    exit_condition=should_continue_loop,  
    description="Iterative refinement with quality threshold"  
)
```

Multi-Agent Loop:

```
# Complex iterative process  
brainstorm_agent = Agent(name="brainstormer", ...)  
designer_agent = Agent(name="designer", ...)  
developer_agent = Agent(name="developer", ...)  
tester_agent = Agent(name="tester", ...)  
  
# Development cycle  
development_loop = LoopAgent(  
    sub_agents=[designer_agent, developer_agent, tester_agent],  
    max_iterations=10,  
    description="Iterative product development cycle"  
)
```

[FLOW] Complex Workflow Composition

| Nested Workflows

Mental Model: Workflows can contain other workflows for hierarchical organization:

```
# Level 1: Individual research tasks
web_agent = Agent(name="web_researcher", ...)
api_agent = Agent(name="api_researcher", ...)
file_agent = Agent(name="file_analyzer", ...)

# Level 2: Parallel research
research_team = ParallelAgent(
    sub_agents=[web_agent, api_agent, file_agent]
)

# Level 3: Sequential processing
processing_pipeline = SequentialAgent(
    sub_agents=[
        research_team,      # Parallel research
        data_cleaner,       # Sequential processing
        analyzer,           # Sequential processing
        reporter            # Sequential processing
    ]
)

# Level 4: Quality loop
quality_assurance = LoopAgent(
    sub_agents=[processing_pipeline, quality_checker, improver],
    max_iterations=3
)
```

| Real-World Example: Content Creation Pipeline

```
# 1. Research Phase (Parallel)
research_sources = ParallelAgent(
    sub_agents=[
        web_research_agent,
        academic_search_agent,
        social_media_monitor
    ]
)

# 2. Content Generation (Sequential)
content_creation = SequentialAgent(
    sub_agents=[
        outline_writer,
        draft_writer,
        fact_checker
    ]
)

# 3. Review & Editing (Loop)
editing_cycle = LoopAgent(
    sub_agents=[
        editor_agent,
        proofreader_agent,
        final_reviewer
    ],
    max_iterations=3
)

# 4. Publication (Sequential)
publication_pipeline = SequentialAgent(
    sub_agents=[
        seo_optimizer,
        formatter_agent,
        publisher_agent
    ]
)

# Complete Content Pipeline
content_workflow = SequentialAgent(
    sub_agents=[
        research_sources,      # Parallel
        content_creation,      # Sequential
        editing_cycle,         # Loop
        publication_pipeline   # Sequential
    ]
)
```

Workflow Decision Framework

| When to Use Each Pattern

Scenario	Sequential	Parallel	Loop
Order matters	✓ Yes	✗ No	✗ No
Independent tasks	✗ No	✓ Yes	✗ No
Need speed	✗ No	✓ Yes	✗ No
Iterative refinement	✗ No	✗ No	✓ Yes
Quality > speed	✗ No	✗ No	✓ Yes
Dependencies	✓ Yes	✗ No	🤔 Maybe

Workflow Selection Guide

Need to orchestrate multiple agents?

- |
- | └─ Steps depend on each other?
 - | | └─ Simple dependency chain?
 - | | | └─ SequentialAgent
 - | | └─ Complex dependencies?
 - | | | └─ SequentialAgent + state routing
- |
- | └─ Steps are independent?
 - | | └─ Need results combined?
 - | | | └─ ParallelAgent + Sequential merger
 - | | └─ Can process separately?
 - | | | └─ ParallelAgent (fire and forget)
- |
- | └─ Need iterative improvement?
 - | | └─ Quality refinement?
 - | | | └─ LoopAgent (critic + refiner)
 - | | └─ Progressive enhancement?
 - | | | └─ LoopAgent (multi-stage improvement)
- |
- | └─ Complex combination?
 - | | └─ Nested workflows (Parallel + Sequential + Loop)

Performance Optimization

Parallel Execution Benefits

Speed Improvements:

- **Independent Tasks:** 3x faster with 3 parallel agents
- **I/O Bound:** Network requests, API calls, file operations
- **CPU Bound:** Distribute across agents with different models

Cost Considerations:

- **Token Efficiency:** Same total tokens, faster execution
- **Model Selection:** Use smaller models for parallel tasks

- **Caching:** Cache intermediate results to avoid recomputation

| Optimization Strategies

Batch Processing:

```
# Process multiple items in parallel
batch_processor = ParallelAgent(
    sub_agents=[
        Agent(name="item_1_processor", ...),
        Agent(name="item_2_processor", ...),
        Agent(name="item_3_processor", ...)
    ]
)

# More efficient than sequential processing
sequential_processor = SequentialAgent(
    sub_agents=[item_1_processor, item_2_processor, item_3_processor]
)
```

Early Exit Optimization:

```
# Stop when good enough
quality_loop = LoopAgent(
    sub_agents=[generator, critic, improver],
    max_iterations=10,
    exit_condition=lambda ctx, res: res.get('quality_score', 0) >= 9
)
```



Debugging Workflows

| State Inspection

Track Data Flow:

```
# Enable state logging
import logging
logging.getLogger('google.adk.agents').setLevel(logging.DEBUG)

# Inspect state at each step
result = await runner.run_async(query)
for event in result.events:
    if 'state' in event:
        print(f"Step: {event.step}")
        print(f"State: {event.state}")
```

Workflow Visualization

Execution Graph:

```
# Generate workflow diagram
workflow_graph = content_pipeline.get_execution_graph()
print(workflow_graph) # Mermaid diagram

# Analyze bottlenecks
performance_report = content_pipeline.analyze_performance()
print(performance_report) # Timing, bottlenecks, optimization suggestions
```

Common Issues & Solutions

Issue	Symptom	Solution
State not passed	Agent can't access previous results	Check <code>output_key</code> and state interpolation
Parallel slowdown	Sequential execution instead of parallel	Verify agents are truly independent
Loop never exits	Infinite refinement cycles	Set <code>max_iterations</code> , add exit conditions
Memory bloat	State growing too large	Use <code>temp:</code> scope, clean up intermediate data
Race conditions	Non-deterministic results	Ensure proper state synchronization

Related Topics


- [Agent Architecture](#) → ([agent-architecture.md](#)): Individual agent design
- [Tools & Capabilities](#) → ([tools-capabilities.md](#)): What agents can do
- [LLM Integration](#) → ([llm-integration.md](#)): How LLMs drive workflows

Hands-On Tutorials

- [Tutorial 04: Sequential Workflows](#) ([04_sequential_workflows.md](#)): Build ordered agent pipelines
- [Tutorial 05: Parallel Processing](#) ([05_parallel_processing.md](#)): Run agents simultaneously for speed
- [Tutorial 06: Multi-Agent Systems](#) ([06_multi_agent_systems.md](#)): Complex agent coordination
- [Tutorial 07: Loop Agents](#) ([07_loop_agents.md](#)): Iterative refinement patterns

Key Takeaways

1. **Sequential:** For ordered, dependent steps (assembly line)
2. **Parallel:** For independent tasks (fan-out/gather)
3. **Loop:** For iterative refinement (critic/refiner pattern)
4. **Composition:** Nest workflows for complex hierarchies
5. **Performance:** Parallel execution for speed, sequential for dependencies
6. **State Flow:** Use `output_key` and interpolation for data passing

 **Next:** Learn about [LLM Integration](#) (`llm-integration.md`) to understand how language models drive these workflows.

Generated on 2025-10-21 09:03:30 from workflows-orchestration.md

Source: Google ADK Training Hub