Agent Architecture & State Management

Description: Understanding agent types, hierarchy patterns, and state vs memory management in ADK

Purpose: Deep dive into agent composition, hierarchy patterns, and state management strategies in ADK.

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

Table of Contents

Agent Types Deep Dive

LLM Agent (Thinker Pattern)

Mental Model: The flexible problem-solver that reasons dynamically:

```
# Basic LLM Agent
agent = Agent(
    name="researcher",
    model="gemini-2.5-flash",
    description="Research and analysis specialist",
    instruction="""
    You are a research assistant who:
    - Analyzes complex topics
    - Uses tools to gather information
    - Provides well-structured answers
    """,
    tools=[search_tool, analysis_tool]
)
```

When to Use:

- ✔️ Conversational interfaces
- ✔️ Creative problem solving
- ✔️ Analysis and reasoning tasks
- ✔️ Dynamic decision making

Characteristics:

- **Flexible**: Adapts to new situations
- **Creative**: Generates novel solutions
- **Tool-Aware**: Can call multiple tools in sequence
- **Stateful**: Maintains conversation context

Workflow Agent (Manager Pattern)

Mental Model: The orchestrator that follows predefined processes:

```python
# Sequential Workflow
sequential_agent = SequentialAgent(
    name="content_pipeline",
    sub_agents=[
        research_agent,    # Step 1: Gather info
        writer_agent,      # Step 2: Create content
        editor_agent       # Step 3: Review & edit
    ],
    description="Complete content creation pipeline"
)

# Parallel Workflow
parallel_agent = ParallelAgent(
    name="data_analyzer",
    sub_agents=[
        stats_agent,       # Analyze numbers
        trends_agent,      # Find patterns
        insights_agent     # Generate insights
    ],
    description="Multi-dimensional data analysis"
)

# Loop Workflow
quality_agent = LoopAgent(
    sub_agents=[
        writer_agent,      # Generate content
        critic_agent       # Evaluate quality
    ],
    max_iterations=3,
    description="Iterative content refinement"
)
```

When to Use:

- ✔️ Predictable, multi-step processes
- ✔️ Quality assurance workflows
- ✔️ Complex orchestration
- ✔️ Batch processing

Remote Agent (Service Pattern)

Mental Model: External specialists accessed via HTTP:

```python
# Remote Agent Integration
youtube_agent = RemoteA2aAgent(
    name='youtube_specialist',
    base_url='https://youtube-agent.example.com',
    description="YouTube content and analytics expert"
)

# Use in local workflow
local_agent = Agent(
    name="content_strategy",
    model="gemini-2.5-flash",
    tools=[AgentTool(youtube_agent)],  # Remote agent as tool
    instruction="Create content strategy using YouTube insights"
)
```

When to Use:

- ✔️ Specialized domain expertise
- ✔️ Microservices architecture
- ✔️ Cross-team collaboration
- ✔️ Scalable agent ecosystems

🏗️ Agent Hierarchy Model

Single Parent Rule

Mental Model: Agents form clean organizational trees like company structures:

```
                    CEO Agent (Root)
                         |
         ┌───────────────┼───────────────┐
         |               |               |
     Research Dept      Analysis Dept    Reporting Dept
    (Sequential)      (ParallelAgent)   (LLMAgent)
         |               |
    ┌────┴────┐     ┌────┼────┐
    |         |     |    |    |
  Fact     Context  Data  Stats Quality
  Finder   Gatherer Team  Team  Check
  (LLM)    (LLM)   (LLM) (LLM) (Loop)
```

Key Rules:

- **Single Parent**: Each agent has exactly one parent
- **Shared State**: Parent and children share session state via `state['key']`
- **Clean Communication**: State keys or `transfer_to_agent()` for handoffs
- **Scoped Execution**: Children inherit parent's invocation context

State Communication Patterns

Pattern 1: State Key Sharing

```
# Parent agent sets context
parent_agent = SequentialAgent(
    sub_agents=[research_agent, analysis_agent],
    description="Research and analyze topic"
)

# Research agent saves to state
research_agent = Agent(
    name="researcher",
    model="gemini-2.5-flash",
    output_key="research_results",  # Saves to state['research_results']
    instruction="Research the topic and save findings"
)

# Analysis agent reads from state
analysis_agent = Agent(
    name="analyst",
    model="gemini-2.5-flash",
    instruction="""
    Analyze the research findings: {research_results}
    Provide insights and recommendations.
    """
)
```

Pattern 2: Agent Tool Integration

```
# Convert agent to tool for another agent
specialist_tool = AgentTool(specialist_agent)

# Use specialist as tool in another agent
orchestrator = Agent(
    name="orchestrator",
    model="gemini-2.5-flash",
    tools=[specialist_tool],
    instruction="Use the expert when you need specialized knowledge"
)
```

Pattern 3: Callback-Based Coordination

```
def on_agent_complete(context, result):
    """Callback when agent finishes"""
    if result.success:
        state['completed_agents'] = state.get('completed_agents', 0) + 1

agent = Agent(
    name="worker",
    model="gemini-2.5-flash",
    instruction="Do your work",
    callbacks=[on_agent_complete]
)
```

💾 State vs Memory Management

Session State (RAM - Short-term)

Mental Model: Working memory for the current conversation:

```
# Session State Examples
state['current_topic'] = "quantum computing"        # Current focus
state['user:name'] = "Alice"                        # User identity
state['temp:calculation'] = intermediate_result     # Temporary data
state['app:version'] = "1.2.3"                      # App-wide data

# State Scopes
state['key']          # Session scope (this conversation)
state['user:key']     # User scope (all user sessions)
state['app:key']      # App scope (entire application)
state['temp:key']     # Temp scope (this invocation only)
```

Characteristics:

- **Scope**: Current session only
- **Persistence**: Lost when session ends
- **Use Cases**: Conversation context, working data, user preferences

- **Performance**: Fast, in-memory access

Memory Service (Hard Drive - Long-term)

Mental Model: Institutional knowledge across all conversations:

```
# Memory Service Integration
runner = Runner(
    memory_service=VertexAiMemoryBankService(
        project="my-project",
        location="us-central1"
    )
)

# Memory automatically captures:
# - Conversation history
# - User patterns
# - Learned facts
# - Important context
```

Characteristics:

- **Scope**: All sessions for user/app
- **Persistence**: Permanent storage
- **Use Cases**: Historical knowledge, user patterns, learned preferences
- **Performance**: Slower, network access

State vs Memory Decision Framework

Scenario	Use State	Use Memory	Example
Current task progress	✔️	❌	`state['step'] = 2`
User identity	✔️	❌	`state['user:name']`
Working calculations	✔️	❌	`state['temp:result']`
Past conversations	❌	✔️	Memory service
Learned preferences	❌	✔️	User behavior patterns
Historical facts	❌	✔️	Important knowledge

Artifacts for Large Content

Mental Model: File system for big data:

```
# Artifact Storage
runner = Runner(
    artifact_service=GcsArtifactService(
        bucket_name="my-agent-artifacts"
    )
)

# Save large content
artifact_id = await runner.save_artifact(
    content=large_report,
    content_type="text/markdown",
    description="Research report"
)

# Reference in state
state['report_id'] = artifact_id
```

When to Use Artifacts:

- Large text documents
- Binary files (images, PDFs)
- Generated content

- Persistent file storage

Agent Communication Patterns

Direct State Transfer

Pattern: Agents communicate through shared state keys:

```
# Agent A produces data
producer_agent = Agent(
    name="data_producer",
    model="gemini-2.5-flash",
    output_key="processed_data",
    instruction="Process the input and save results to state"
)

# Agent B consumes data
consumer_agent = Agent(
    name="data_consumer",
    model="gemini-2.5-flash",
    instruction="""
    Use this data: {processed_data}
    Generate final output.
    """
)
```

Tool-Based Communication

Pattern: Agents call each other as tools:

```python
# Specialist agent
expert_agent = Agent(
    name="domain_expert",
    model="gemini-2.5-pro",
    instruction="You are a specialist in quantum physics"
)

# General agent uses specialist
general_agent = Agent(
    name="general_assistant",
    model="gemini-2.5-flash",
    tools=[AgentTool(expert_agent)],
    instruction="Use the expert when you need specialized knowledge"
)
```

Callback-Based Coordination

Pattern: Use callbacks for cross-agent coordination:

```python
def on_agent_complete(context, result):
    """Callback when agent finishes"""
    if result.success:
        state['completed_agents'] = state.get('completed_agents', 0) + 1

agent = Agent(
    name="worker",
    model="gemini-2.5-flash",
    instruction="Do your work",
    callbacks=[on_agent_complete]
)
```

🏭 Composition Best Practices

Hierarchical Design Principles

1. **Single Responsibility**: Each agent has one clear purpose
2. **Clean Interfaces**: Communicate through state keys, not direct calls

3. **Scoped State**: Use appropriate state prefixes (`temp:`, `user:`, `app:`)
4. **Error Boundaries**: Isolate failures within agent boundaries

Performance Optimization

```
# Parallel when possible
parallel_workflow = ParallelAgent(
    sub_agents=[
        fast_task_1,    # Independent
        fast_task_2,    # Independent
        fast_task_3     # Independent
    ]
)

# Sequential when dependent
sequential_workflow = SequentialAgent(
    sub_agents=[
        setup_agent,    # Must run first
        process_agent,  # Needs setup results
        finish_agent    # Needs process results
    ]
)
```

Error Handling Patterns

```
# Graceful degradation
robust_agent = Agent(
    name="robust_worker",
    model="gemini-2.5-flash",
    instruction="""
    Try to complete the task.
    If you encounter errors, save error details to state
    and provide fallback response.
    """,
    output_key="result"
)

# Circuit breaker pattern
def circuit_breaker_callback(context, result):
    failure_count = state.get('failures', 0) + 1
    state['failures'] = failure_count
    if failure_count > 3:
        # Stop trying
        context.cancel()
```

🔍 Debugging Agent Systems

State Inspection

```
# Debug state during execution
debug_agent = Agent(
    name="debugger",
    model="gemini-2.5-flash",
    instruction="""
    Current state: {debug:state}
    Available keys: {debug:keys}
    Analyze the current situation.
    """,
    tools=[debug_tool]
)
```

Event Tracing

```
# Enable detailed event logging
runner = Runner(
    event_service=LoggingEventService(
        level="DEBUG",
        include_state=True
    )
)

# Events captured:
# - AGENT_START
# - TOOL_CALL_START
# - LLM_REQUEST
# - STATE_CHANGE
# - AGENT_COMPLETE
```

Common Issues & Solutions

Issue	Symptom	Solution
State not shared	Agent can't see previous results	Use `output_key` and state interpolation
Memory leaks	State growing indefinitely	Use `temp:` prefix for temporary data
Circular dependencies	Agents waiting for each other	Redesign hierarchy, use ParallelAgent
Performance degradation	Slow response times	Add ParallelAgent, reduce state size

---

📚 Related Topics

- **[Tools & Capabilities →](tools-capabilities.md)**: Extend agent capabilities
- **[Workflows & Orchestration →](workflows-orchestration.md)**: Advanced composition patterns

- **Production & Deployment → (production-deployment.md)**: Running agent systems at scale

---

🎯 Key Takeaways

1. **Agent Types**: Choose LLM for flexibility, Workflow for process, Remote for specialization
2. **Hierarchy**: Single parent rule, clean state communication
3. **State Management**: Session for current work, Memory for long-term knowledge
4. **Communication**: State keys for sharing, AgentTool for integration
5. **Best Practices**: Single responsibility, clean interfaces, error boundaries

🔗 **Next**: Learn about Tools & Capabilities (tools-capabilities.md) to extend what your agents can do.

Generated on 2025-10-21 09:03:14 from agent-architecture.md

Source: Google ADK Training Hub