Google Agent Development Kit (ADK) - Ultimate Reference Manual

Overview and Introduction

What is Google Agent Development Kit (ADK)?

Google Agent Development Kit (ADK) is a **flexible and modular framework** for developing and deploying AI agents[1] [2]. While optimized for Gemini and the Google ecosystem, ADK is **model-agnostic**, **deployment-agnostic**, and built for **compatibility with other frameworks**. It's designed to make agent development feel more like traditional software development[1] [3].

Key Characteristics:

- **Multi-agent by design**: Compose agents in parallel, sequential, or hierarchical workflows[4]
- **Model-agnostic**: Works with Gemini, GPT-4o, Claude, Mistral, and others via LiteLLM[4]
- **Production-ready**: Same framework powering Google products like Agentspace and Customer Engagement Suite[2]
- **Deployment flexible**: Run locally, scale with Vertex AI Agent Engine, or integrate into custom infrastructure[1]

Core Agent Architecture

Agent Categories and Types

ADK provides three distinct agent categories to build sophisticated applications[5]:

1. LLM Agents (`LlmAgent`, `Agent`)

Purpose: Utilize Large Language Models as their core engine for understanding natural language, reasoning, planning, and dynamic decision-making.

Core Configuration Parameters:

```
from google.adk.agents import LlmAgent

agent = LlmAgent(
    model="gemini-2.0-flash",           # Required: LLM identifier
    name="unique_agent_name",           # Required: Unique string identifier
    description="Agent description",     # Optional: Capability summary
    instruction="System prompt text",    # Core behavior guidance
    tools=[tool1, tool2],               # Optional: List of tools
    generate_content_config=config,     # Optional: LLM generation parameters
    input_schema=InputSchema,           # Optional: Input validation schema
    output_schema=OutputSchema,         # Optional: Output structure schema
    output_key="result_key",            # Optional: State storage key
    include_contents='default',         # Optional: Context control
    planner=planner_instance,           # Optional: Planning strategy
```

```
    code_executor=executor,             # Optional: Code execution capability
)
```

Key Parameters Explained:

- `name` (Required): Unique string identifier crucial for multi-agent systems where agents refer to each other. Choose descriptive names (e.g., `customer_support_router`, `billing_inquiry_agent`). Avoid reserved names like `user`.
- `instruction` (Critical): The most important parameter for shaping behavior. Should include:
  - Core task or goal
  - Personality or persona
  - Behavioral constraints
  - Tool usage guidance
  - Output format requirements
  - Can use template syntax: `{var}` for state variables, `{artifact.var}` for artifacts, `{var?}` to ignore missing variables
- `tools`: List of capabilities including:
  - Native functions (automatically wrapped as `FunctionTool`)
  - `BaseTool` instances
  - Other agents (`AgentTool`)
  - Built-in tools (Search, Code Exec)

Advanced Configuration:

Structured Data Handling:

- `input_schema`: Defines expected input structure; user content must be JSON conforming to this schema
- `output_schema`: Defines desired output structure; **disables tool usage and agent transfers**
- `output_key`: Automatically saves agent's final response to session state under this key

Planning Capabilities:

- `BuiltInPlanner`: Leverages model's built-in planning (e.g., Gemini's thinking feature)

```
from google.adk.planners import BuiltInPlanner
from google.genai import types

planner = BuiltInPlanner(
    thinking_config=types.ThinkingConfig(
        include_thoughts=True,
        thinking_budget=1024
    )
)
```

- `PlanReActPlanner`: Structured plan-reason-act approach for models without built-in thinking

```
from google.adk.planners import PlanReActPlanner
planner = PlanReActPlanner()
```

2. Workflow Agents (`SequentialAgent`, `ParallelAgent`, `LoopAgent`)

Purpose: Control execution flow of sub-agents using predefined, deterministic patterns without LLM involvement for orchestration.

Sequential Agent: Executes sub-agents one after another in sequence

```
from google.adk.agents import SequentialAgent

sequential_agent = SequentialAgent(
    name="workflow_sequential",
    sub_agents=[agent1, agent2, agent3],
    description="Processes tasks in order"
)
```

Parallel Agent: Executes multiple sub-agents simultaneously

```
from google.adk.agents import ParallelAgent

parallel_agent = ParallelAgent(
    name="workflow_parallel",
    sub_agents=[agent1, agent2, agent3],
    description="Processes tasks in parallel"
)
```

Loop Agent: Repeatedly executes sub-agents until termination condition

```
from google.adk.agents import LoopAgent

loop_agent = LoopAgent(
    name="workflow_loop",
    sub_agents=[agent1, agent2],
    max_iterations=10,
    description="Repeats until condition met"
)
```

3. Custom Agents (extending `BaseAgent`)

Purpose: Implement unique operational logic, specialized integrations, or control flows not covered by standard types.

```
from google.adk.agents import BaseAgent

class CustomAgent(BaseAgent):
    def __init__(self, name: str, **kwargs):
        super().__init__(name=name, **kwargs)
```

```
    async def _run_async_impl(self, new_message, context):
        # Custom agent logic here
        pass
```

Multi-Agent Systems and Hierarchies

Agent Composition Primitives

Agent Hierarchy (Parent-Child Relationships)

ADK enables tree structures through parent-child relationships defined in `BaseAgent`[6]:

```
# Establishing hierarchy
parent_agent = LlmAgent(
    name="parent_agent",
    model="gemini-2.0-flash",
    sub_agents=[child_agent1, child_agent2]  # Pass list of agent instances
)

# Accessing hierarchy
children = parent_agent.sub_agents
parent = child_agent1.parent_agent
```

Agent Transfer Mechanisms

Dynamic Transfer: Agents can transfer control to other agents using:

1. **LLM-driven transfer**: Via instruction and reasoning
2. **Tool-driven transfer**: Using `tool_context.actions.transfer_to_agent`
3. **Explicit transfer**: Direct API calls

```
# Tool-driven transfer example
def escalate_to_specialist(query: str, tool_context: ToolContext) -> str:
    if "urgent" in query.lower():
        tool_context.actions.transfer_to_agent = "specialist_agent"
        return "Transferring to specialist..."
    return "Handling normally"
```

Multi-Agent Coordination Patterns

Controller Pattern

Central agent delegates to specialized sub-agents:

```
host_agent = LlmAgent(
    name="host_agent",
```

```
    instruction="Coordinate tasks by calling specialized agents",
    sub_agents=[flight_agent, hotel_agent, activity_agent]
)
```

Hierarchical Specialist Pattern

Nested specialization with escalation paths:

```
support_agent = LlmAgent(
    name="tier1_support",
    sub_agents=[technical_expert, billing_expert],
    instruction="Handle basic queries, escalate complex issues"
)
```

Tools and Capabilities System

Tool Types and Categories

1. Function Tools (Custom Tools)

Definition: Custom capabilities created by developers for specific application needs.

Function Tool Creation:

```
def get_weather(city: str) -> dict:
    """Retrieves current weather for specified city.

    Args:
        city: Name of the city

    Returns:
        dict: Weather information with status and data
    """
    # Implementation here
    return {"status": "success", "temperature": "25°C"}

# Usage in agent
agent = LlmAgent(
    name="weather_agent",
    tools=[get_weather],  # Auto-wrapped as FunctionTool
    instruction="Use get_weather tool to provide weather information"
)
```

Advanced Function Tools with ToolContext:

```
from google.adk.tools import ToolContext

def advanced_tool(param: str, tool_context: ToolContext) -> dict:
    """Advanced tool with context access."""
    # Access session state
```

```
    user_prefs = tool_context.state.get("user:preferences", {})

    # Modify state
    tool_context.state["last_query"] = param

    # Control agent flow
    if "urgent" in param:
        tool_context.actions.transfer_to_agent = "urgent_handler"

    # Access artifacts and memory
    artifacts = tool_context.list_artifacts()
    memory_results = tool_context.search_memory(param)

    return {"status": "processed", "param": param}
```

Tool Function Best Practices:

- **Naming**: Use descriptive verb-noun names (`get_weather`, `schedule_meeting`)
- **Parameters**:
  - Clear, descriptive names
  - Type hints required (Python)
  - JSON-serializable types only
  - No default values
- **Returns**: Must be dictionary/Map
- **Docstrings**: Critical for LLM understanding
  - State what tool does
  - Explain when to use
  - Describe parameters
  - Explain return structure

2. Built-in Tools

ADK provides ready-to-use tools for common tasks:

Google Search Tool:

```
from google.adk.tools import google_search

agent = LlmAgent(
    name="search_agent",
    tools=[google_search],
    instruction="Use google_search to find current information"
)
```

Code Execution Tool:

```
from google.adk.tools import code_executor

agent = LlmAgent(
    name="code_agent",
    code_executor=code_executor,
    instruction="You can execute code to solve problems"
)
```

3. Agent-as-Tool

Use specialized agents as tools for other agents:

```
from google.adk.tools import AgentTool

specialist_tool = AgentTool(specialist_agent)
generalist_agent = LlmAgent(
    name="generalist",
    tools=[specialist_tool],
    instruction="Delegate complex tasks to specialist"
)
```

4. Third-Party Tool Integration

LangChain Tools:

```
from langchain.tools import SomeToolClass
from google.adk.tools import LangChainTool

langchain_tool = LangChainTool(SomeToolClass())
agent = LlmAgent(tools=[langchain_tool])
```

Tool Context and Advanced Capabilities

ToolContext Components

The `ToolContext` provides tools with rich contextual information and control:

State Management:

```
def state_aware_tool(param: str, tool_context: ToolContext) -> dict:
    # Read state (dictionary-like access)
    current_value = tool_context.state.get("key", default)

    # Write state (changes tracked and persisted)
    tool_context.state["new_key"] = "new_value"

    # State prefixes
    tool_context.state["app:global_setting"] = "value"  # App-wide
    tool_context.state["user:preference"] = "value"     # User-specific
```

```
    tool_context.state["temp:session_data"] = "value"   # Temporary

    return {"status": "updated"}
```

Agent Flow Control:

```
def flow_control_tool(command: str, tool_context: ToolContext) -> dict:
    # Skip LLM summarization
    tool_context.actions.skip_summarization = True

    # Transfer to another agent
    tool_context.actions.transfer_to_agent = "specialist_agent"

    # Escalate to parent agent
    tool_context.actions.escalate = True

    return {"message": "Action completed"}
```

Artifact Management:

```
def artifact_tool(filename: str, tool_context: ToolContext) -> dict:
    # List available artifacts
    artifacts = tool_context.list_artifacts()

    # Load specific artifact
    content = tool_context.load_artifact(filename)

    # Save new artifact
    from google.genai import types
    new_artifact = types.Part.from_text("New content")
    version = tool_context.save_artifact("new_file.txt", new_artifact)

    return {"artifacts": len(artifacts), "new_version": version}
```

Memory Integration:

```
def memory_aware_tool(query: str, tool_context: ToolContext) -> dict:
    # Search long-term memory
    memory_results = tool_context.search_memory(query)

    relevant_info = []
    for memory in memory_results.memories:
        if memory.events:
            text = memory.events[^0].content.parts[^0].text
            relevant_info.append(text)

    return {"query": query, "memory_context": relevant_info}
```

State Management and Sessions

Session Architecture

Core Session Concepts

- **Session**: Single, ongoing interaction between user and agent system containing chronological Events
- **State**: Data within specific session (`session.state`)
- **Memory**: Cross-session, searchable information store

Session Lifecycle Management

```
from google.adk.sessions import InMemorySessionService

# Create session service
session_service = InMemorySessionService()

# Create new session
session = await session_service.create_session(
    app_name="my_app",
    user_id="user123",
    session_id="session456"
)

# Get existing session
existing_session = await session_service.get_session(
    app_name="my_app",
    user_id="user123",
    session_id="session456"
)
```

State Management Patterns

State Prefixes and Scope:

- **No prefix**: Session-specific data
- `app:`: Application-wide data (all users)
- `user:`: User-specific data (across sessions)
- `temp:`: Temporary data (not persisted)

```
# State manipulation examples
session.state["current_step"] = "processing"         # Session scope
session.state["user:preferences"] = {"theme": "dark"} # User scope
session.state["app:version"] = "1.0.0"               # App scope
session.state["temp:calculation"] = 42               # Temporary
```

SessionService Implementations

In-Memory (Development):

```
from google.adk.sessions import InMemorySessionService
session_service = InMemorySessionService()
# Data lost on restart - development only
```

Cloud-Based (Production):

```
# Production implementations for persistence
# (Specific implementations depend on deployment target)
```

Memory Architecture

Memory Service Capabilities

```
from google.adk.memory import MemoryService

# Search memory across sessions
memory_results = await memory_service.search_memory(
    user_id="user123",
    query="previous hotel preferences"
)

# Process results
for memory in memory_results.memories:
    for event in memory.events:
        content = event.content.parts[^0].text
        # Use relevant historical context
```

Event System and Callbacks

Event Architecture

ADK operates on an event-driven architecture where all interactions generate Events that flow through the system.

Event Types

- **User events**: Messages from users
- **Model events**: LLM responses
- **Tool events**: Tool executions
- **Agent events**: Agent state changes
- **Final response events**: Completed interactions

Callback System

Callback Types and Purposes

Callbacks provide hooks into agent execution at specific points:

1. Agent-Level Callbacks:

```
def before_agent_callback(callback_context: CallbackContext, new_message: types.Content)
    """Called before agent's main execution logic."""
    print(f"Agent {callback_context.agent_name} starting work")
    # Return None: proceed normally
    # Return Content: skip agent execution, use returned content
    return None

def after_agent_callback(callback_context: CallbackContext, agent_response: types.Content
    """Called after agent completes execution."""
    # Modify or replace agent's response
    return agent_response

agent = LlmAgent(
    name="callback_agent",
    before_agent_callback=before_agent_callback,
    after_agent_callback=after_agent_callback
)
```

2. Model-Level Callbacks:

```
def before_model_callback(callback_context: CallbackContext, llm_request: LlmRequest) ->
    """Called before LLM request."""
    # Inspect/modify request
    print(f"Sending to LLM: {llm_request}")

    # Implement guardrails
    if "blocked_content" in llm_request.contents[-1].parts[^0].text:
        return LlmResponse(
            content=types.Content(
                role="model",
                parts=[types.Part(text="Content blocked by policy")]
            )
        )

    # Modify system instruction
    instruction = llm_request.config.system_instruction
    if instruction and instruction.parts:
        modified_text = "[MODIFIED] " + instruction.parts[^0].text
        instruction.parts[^0].text = modified_text

    return None  # Proceed with (potentially modified) request

def after_model_callback(callback_context: CallbackContext, llm_response: LlmResponse) ->
    """Called after LLM responds."""
    # Post-process response
```

```
    content = llm_response.content
    if content and content.parts:
        modified_text = content.parts[^0].text + "\n[Processed by callback]"
        content.parts[^0].text = modified_text

    return llm_response

agent = LlmAgent(
    name="model_callback_agent",
    before_model_callback=before_model_callback,
    after_model_callback=after_model_callback
)
```

3. Tool-Level Callbacks:

```
def before_tool_callback(callback_context: CallbackContext, tool_name: str, tool_args: di
    """Called before tool execution."""
    print(f"Executing tool: {tool_name} with args: {tool_args}")

    # Validate arguments
    if tool_name == "sensitive_operation" and not callback_context.state.get("user:author
        return {"error": "Unauthorized access"}

    return None  # Proceed with tool execution

def after_tool_callback(callback_context: CallbackContext, tool_name: str, tool_result: d
    """Called after tool execution."""
    # Log tool results
    print(f"Tool {tool_name} returned: {tool_result}")

    # Sanitize sensitive data
    if "password" in tool_result:
        tool_result["password"] = "[REDACTED]"

    return tool_result

agent = LlmAgent(
    name="tool_callback_agent",
    tools=[my_tool],
    before_tool_callback=before_tool_callback,
    after_tool_callback=after_tool_callback
)
```

CallbackContext Components

```
def comprehensive_callback(callback_context: CallbackContext, *args):
    # Agent information
    agent_name = callback_context.agent_name

    # Session access
    current_state = callback_context.state
    current_state["callback_executed"] = True
```

```
    # Service access (if configured)
    artifacts = callback_context.list_artifacts()
    memory_results = callback_context.search_memory("relevant query")

    # Event tracking
    event_id = callback_context.event_id
```

Models and Integrations

Model Configuration

ADK supports multiple model providers through different integration approaches:

Google Models (Direct Integration)

```
# Gemini models via direct integration
agent = LlmAgent(
    model="gemini-2.0-flash",          # Fast, efficient
    model="gemini-2.0-flash-exp",     # Experimental features
    model="gemini-2.5-pro",           # Advanced reasoning
    model="gemini-2.5-flash"          # Balanced performance
)
```

LiteLLM Integration (Multi-Provider)

```
from google.adk.models.lite_llm import LiteLlm

# OpenAI models
openai_model = LiteLlm("openai/gpt-4")

# Anthropic models
claude_model = LiteLlm("anthropic/claude-3-5-sonnet-20241022")

# Other providers
mistral_model = LiteLlm("mistral/mistral-large-latest")

agent = LlmAgent(
    name="multi_provider_agent",
    model=openai_model,  # Use LiteLlm wrapper
    instruction="Work with any supported model"
)
```

Model Configuration Parameters

```
from google.genai import types

# Advanced model configuration
config = types.GenerateContentConfig(
    temperature=0.2,           # Lower = more deterministic
    max_output_tokens=1024,    # Response length limit
```

```
    top_p=0.8,                # Nucleus sampling
    top_k=40,                 # Top-k sampling
    candidate_count=1,        # Number of responses
    stop_sequences=["END"]    # Stop generation tokens
)

agent = LlmAgent(
    model="gemini-2.0-flash",
    generate_content_config=config
)
```

Runner System and Execution

Runner Architecture

The Runner system manages agent execution, session handling, and event streaming.

Basic Runner Setup

```
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService

# Create session service
session_service = InMemorySessionService()

# Create runner
runner = Runner(
    agent=my_agent,
    app_name="my_application",
    session_service=session_service
)
```

Executing Agents

Asynchronous Execution:

```
from google.genai import types

async def run_agent_interaction():
    # Create user message
    user_message = types.Content(
        role='user',
        parts=[types.Part(text="Hello, help me with a task")]
    )

    # Run agent and stream events
    async for event in runner.run_async(
        user_id="user123",
        session_id="session456",
        new_message=user_message
    ):
```

```
        # Process events as they arrive
        if event.is_final_response():
            final_response = event.content.parts[^0].text
            print(f"Agent response: {final_response}")
        elif event.is_tool_call():
            print(f"Tool called: {event.tool_name}")
```

Synchronous Execution:

```
# For simpler use cases
response = await runner.run_once(
    user_id="user123",
    session_id="session456",
    message="Process this request"
)
```

Event Stream Processing

```
async def process_event_stream(runner, user_id, session_id, message):
    events = runner.run_async(user_id, session_id, message)

    async for event in events:
        event_type = event.type

        if event.is_user_message():
            print("User input received")
        elif event.is_model_request():
            print("Sending request to LLM")
        elif event.is_model_response():
            print("Received LLM response")
        elif event.is_tool_call():
            print(f"Calling tool: {event.tool_name}")
        elif event.is_tool_response():
            print(f"Tool result: {event.tool_result}")
        elif event.is_agent_transfer():
            print(f"Transferring to: {event.target_agent}")
        elif event.is_final_response():
            print(f"Final answer: {event.content}")
            break
```

Deployment and Production

Local Development

Development Server:

```
# Install ADK
pip install google-adk

# Create new project
adk create my_agent_app
```

```
# Run development server
adk web
```

CLI Commands:

```
adk api          # Start API server
adk web          # Start web UI
adk eval         # Run evaluations
adk deploy       # Deploy to cloud
```

Production Deployment

Containerization

```
FROM python:3.11-slim

WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt

COPY . .
EXPOSE 8080

CMD ["adk", "api", "--port", "8080"]
```

Google Cloud Deployment

Vertex AI Agent Engine:

```
# Deploy to Vertex AI
from google.adk.deployment import VertexAIDeployment

deployment = VertexAIDeployment(
    agent=my_agent,
    project_id="my-gcp-project",
    region="us-central1"
)

endpoint = await deployment.deploy()
```

Cloud Run Deployment:

```
# cloud-run-config.yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: adk-agent-service
spec:
```

```
  template:
    spec:
      containers:
      - image: gcr.io/project/adk-agent:latest
        ports:
        - containerPort: 8080
        env:
        - name: MODEL_NAME
          value: "gemini-2.0-flash"
```

Agent2Agent (A2A) Protocol

A2A Protocol Overview

The Agent2Agent protocol enables standardized communication between agents across different platforms and frameworks[4].

A2A Implementation

Agent Metadata (`.well-known/agent.json`):

```
{
  "name": "specialist_agent",
  "description": "Handles specialized technical queries",
  "version": "1.0.0",
  "endpoints": {
    "run": "/run"
  },
  "schema": {
    "input": {...},
    "output": {...}
  }
}
```

A2A Server Setup:

```
from fastapi import FastAPI
from google.adk.a2a import A2AServer

app = FastAPI()
a2a_server = A2AServer(agent=my_agent)

@app.post("/run")
async def run_agent(request: dict):
    return await a2a_server.handle_request(request)

# Automatic A2A metadata endpoint
@app.get("/.well-known/agent.json")
async def agent_metadata():
    return a2a_server.get_metadata()
```

A2A Client Usage:

```
import httpx

async def call_remote_agent(agent_url: str, payload: dict):
    async with httpx.AsyncClient() as client:
        response = await client.post(f"{agent_url}/run", json=payload)
        return response.json()

# Use in agent tool
def call_specialist(query: str) -> dict:
    """Call remote specialist agent via A2A."""
    payload = {"query": query, "context": "technical"}
    result = await call_remote_agent("https://specialist.example.com", payload)
    return result
```

Model Context Protocol (MCP)

MCP Integration

ADK supports Model Context Protocol for standardized tool interfaces.

```
from google.adk.tools.mcp import MCPTool

# Connect to MCP server
mcp_tool = MCPTool(
    server_url="stdio://mcp-server-command",
    tool_name="filesystem_operations"
)

agent = LlmAgent(
    name="mcp_agent",
    tools=[mcp_tool],
    instruction="Use MCP tools for file operations"
)
```

Evaluation and Testing

Built-in Evaluation Framework

```
from google.adk.evaluation import Evaluator, TestCase

# Define test cases
test_cases = [
    TestCase(
        input="What's the weather in Paris?",
        expected_output_contains=["Paris", "weather"],
        expected_tools_called=["get_weather"]
    ),
    TestCase(
```

```
        input="Book a flight to Tokyo",
        expected_agent_transfer="booking_specialist"
    )
]

# Run evaluation
evaluator = Evaluator(agent=my_agent)
results = await evaluator.evaluate(test_cases)

# Analyze results
for result in results:
    print(f"Test: {result.input}")
    print(f"Passed: {result.passed}")
    print(f"Score: {result.score}")
```

Custom Evaluation Metrics

```
def custom_evaluation_metric(response: str, expected: str) -> float:
    """Custom scoring logic."""
    # Implement your evaluation criteria
    return similarity_score(response, expected)

evaluator = Evaluator(
    agent=my_agent,
    custom_metrics=[custom_evaluation_metric]
)
```

Security and Safety Patterns

Input Validation and Sanitization

```
def input_validator(callback_context: CallbackContext, new_message: types.Content) -> Opt
    """Validate and sanitize user input."""
    user_text = new_message.parts[^0].text

    # Check for malicious patterns
    if contains_injection_attempt(user_text):
        return types.Content(
            role="model",
            parts=[types.Part(text="Invalid input detected")]
        )

    # Sanitize input
    sanitized_text = sanitize_input(user_text)
    new_message.parts[^0].text = sanitized_text

    return None  # Proceed with sanitized input

secure_agent = LlmAgent(
    name="secure_agent",
```

```
    before_agent_callback=input_validator
)
```

Output Filtering

```
def output_filter(callback_context: CallbackContext, llm_response: LlmResponse) -> Optior
    """Filter sensitive information from responses."""
    response_text = llm_response.content.parts[^0].text

    # Remove sensitive patterns
    filtered_text = remove_sensitive_info(response_text)

    # Update response
    llm_response.content.parts[^0].text = filtered_text
    return llm_response

secure_agent = LlmAgent(
    name="filtered_agent",
    after_model_callback=output_filter
)
```

Access Control

```
def access_control(callback_context: CallbackContext, tool_name: str, tool_args: dict) ->
    """Implement role-based access control."""
    user_role = callback_context.state.get("user:role", "guest")

    restricted_tools = ["admin_operation", "sensitive_data_access"]

    if tool_name in restricted_tools and user_role != "admin":
        return {"error": "Access denied", "required_role": "admin"}

    return None  # Allow tool execution

secure_agent = LlmAgent(
    name="access_controlled_agent",
    tools=[sensitive_tool],
    before_tool_callback=access_control
)
```

Advanced Patterns and Best Practices

Agent Composition Patterns

Specialist Router Pattern

```
router_agent = LlmAgent(
    name="intelligent_router",
    instruction="""
    Analyze user queries and route to appropriate specialists:
    - Technical issues → technical_support_agent
    - Billing questions → billing_agent
    - General inquiries → general_support_agent
    """,
    sub_agents=[technical_agent, billing_agent, general_agent]
)
```

Pipeline Pattern

```
pipeline_agent = SequentialAgent(
    name="processing_pipeline",
    sub_agents=[
        input_validator_agent,
        data_processor_agent,
        output_formatter_agent
    ]
)
```

Consensus Pattern

```
consensus_agent = ParallelAgent(
    name="consensus_system",
    sub_agents=[expert1, expert2, expert3],
    post_processor=consensus_aggregator
)
```

Error Handling and Resilience

Retry Logic

```
def retry_on_failure(callback_context: CallbackContext, llm_response: LlmResponse) -> Opt
    """Implement retry logic for failed responses."""
    if is_error_response(llm_response):
        retry_count = callback_context.state.get("retry_count", 0)

        if retry_count < 3:
            callback_context.state["retry_count"] = retry_count + 1
            # Trigger retry by returning None and modifying request
            return None
        else:
            return create_fallback_response()
```

```
    return llm_response
```

Circuit Breaker Pattern

```
class CircuitBreakerTool:
    def __init__(self, wrapped_tool, failure_threshold=5):
        self.wrapped_tool = wrapped_tool
        self.failure_count = 0
        self.failure_threshold = failure_threshold
        self.circuit_open = False

    def __call__(self, *args, tool_context: ToolContext = None, **kwargs):
        if self.circuit_open:
            return {"error": "Circuit breaker open", "status": "unavailable"}

        try:
            result = self.wrapped_tool(*args, **kwargs)
            self.failure_count = 0  # Reset on success
            return result
        except Exception as e:
            self.failure_count += 1
            if self.failure_count >= self.failure_threshold:
                self.circuit_open = True
            raise e
```

Performance Optimization

Caching Strategies

```
def caching_callback(callback_context: CallbackContext, llm_request: LlmRequest) -> Optic
    """Implement response caching."""
    request_hash = hash_request(llm_request)
    cached_response = callback_context.state.get(f"cache:{request_hash}")

    if cached_response:
        return LlmResponse.from_dict(cached_response)

    return None  # Proceed with LLM call

def cache_response(callback_context: CallbackContext, llm_response: LlmResponse) -> Optic
    """Cache successful responses."""
    if llm_response.success:
        request_hash = hash_current_request(callback_context)
        callback_context.state[f"cache:{request_hash}"] = llm_response.to_dict()

    return llm_response
```

Async Tool Execution

```
import asyncio

async def parallel_tool_execution(queries: list[str], tool_context: ToolContext) -> dict:
    """Execute multiple tools in parallel."""
    tasks = []
    for query in queries:
        task = asyncio.create_task(async_tool_call(query))
        tasks.append(task)

    results = await asyncio.gather(*tasks)
    return {"results": results, "count": len(results)}
```

Complete Example: Multi-Agent Travel Assistant

This comprehensive example demonstrates most ADK concepts in a real-world scenario:

```
from google.adk.agents import LlmAgent, SequentialAgent
from google.adk.tools import FunctionTool, ToolContext
from google.adk.sessions import InMemorySessionService
from google.adk.runners import Runner
from google.genai import types
import asyncio

# --- Tools ---
def search_flights(origin: str, destination: str, date: str, tool_context: ToolContext) -
    """Search for available flights."""
    # Store search parameters
    tool_context.state["flight_search"] = {
        "origin": origin, "destination": destination, "date": date
    }

    # Simulate flight search
    flights = [
        {"airline": "Example Air", "price": 299, "duration": "5h 30m"},
        {"airline": "Sky Lines", "price": 349, "duration": "4h 45m"}
    ]

    return {"status": "success", "flights": flights, "count": len(flights)}

def book_hotel(location: str, checkin: str, checkout: str, tool_context: ToolContext) ->
    """Search and book hotel accommodations."""
    # Access previous flight search for context
    flight_info = tool_context.state.get("flight_search", {})

    hotels = [
        {"name": "Grand Hotel", "price": 150, "rating": 4.5},
        {"name": "Budget Inn", "price": 89, "rating": 3.8}
    ]

    # Store booking preference
    tool_context.state["hotel_search"] = {
        "location": location, "checkin": checkin, "checkout": checkout
```

```
    }

    return {"status": "success", "hotels": hotels}

def get_activities(destination: str, interests: list[str], tool_context: ToolContext) ->
    """Find activities and attractions."""
    activities = [
        {"name": "City Museum", "type": "culture", "price": 15},
        {"name": "Food Tour", "type": "food", "price": 45},
        {"name": "Harbor Cruise", "type": "sightseeing", "price": 30}
    ]

    # Filter by interests
    if interests:
        filtered_activities = [a for a in activities if a["type"] in interests]
    else:
        filtered_activities = activities

    return {"status": "success", "activities": filtered_activities}

# --- Specialized Agents ---
flight_agent = LlmAgent(
    name="flight_specialist",
    model="gemini-2.0-flash",
    description="Specializes in flight search and booking",
    instruction="""
    You are a flight booking specialist. Use the search_flights tool to find flights.
    Always ask for origin, destination, and travel date if not provided.
    Present options clearly with prices and durations.
    """,
    tools=[search_flights],
    output_key="flight_results"
)

hotel_agent = LlmAgent(
    name="hotel_specialist",
    model="gemini-2.0-flash",
    description="Handles hotel search and recommendations",
    instruction="""
    You are a hotel booking specialist. Use book_hotel tool to find accommodations.
    Consider the flight destination from previous searches: {flight_search.destination?}
    Ask for check-in and check-out dates if needed.
    """,
    tools=[book_hotel],
    output_key="hotel_results"
)

activity_agent = LlmAgent(
    name="activity_specialist",
    model="gemini-2.0-flash",
    description="Recommends activities and attractions",
    instruction="""
    You are an activities specialist. Use get_activities to find things to do.
    Consider the destination: {flight_search.destination?}
    Ask about interests (culture, food, sightseeing, adventure) to personalize recommenda
    """,
```

```
    tools=[get_activities],
    output_key="activity_results"
)

# --- Coordination Agent ---
def transfer_to_specialist(query: str, tool_context: ToolContext) -> dict:
    """Route queries to appropriate specialists."""
    query_lower = query.lower()

    if any(word in query_lower for word in ["flight", "fly", "airline", "departure"]):
        tool_context.actions.transfer_to_agent = "flight_specialist"
        return {"routing": "flight_specialist", "reason": "Flight-related query"}
    elif any(word in query_lower for word in ["hotel", "accommodation", "stay", "room"]):
        tool_context.actions.transfer_to_agent = "hotel_specialist"
        return {"routing": "hotel_specialist", "reason": "Hotel-related query"}
    elif any(word in query_lower for word in ["activity", "attraction", "things to do", '
        tool_context.actions.transfer_to_agent = "activity_specialist"
        return {"routing": "activity_specialist", "reason": "Activity-related query"}
    else:
        return {"routing": "continue", "reason": "General travel planning query"}

coordinator_agent = LlmAgent(
    name="travel_coordinator",
    model="gemini-2.0-flash",
    description="Coordinates travel planning across specialists",
    instruction="""
    You are a travel planning coordinator. Help users plan complete trips.

    Use transfer_to_specialist tool to route specific requests:
    - Flight queries → flight_specialist
    - Hotel queries → hotel_specialist
    - Activity queries → activity_specialist

    For general planning, gather requirements then coordinate with specialists.
    Summarize results from state:
    - Flights: {flight_results?}
    - Hotels: {hotel_results?}
    - Activities: {activity_results?}
    """,
    tools=[transfer_to_specialist],
    sub_agents=[flight_agent, hotel_agent, activity_agent]
)

# --- Callbacks for Monitoring ---
def log_agent_activity(callback_context, new_message):
    """Log all agent interactions."""
    print(f"[{callback_context.agent_name}] Processing: {new_message.parts[^0].text[:50]}
    return None

def log_tool_usage(callback_context, tool_name, tool_args):
    """Log tool executions."""
    print(f"[{callback_context.agent_name}] Using tool: {tool_name}")
    return None

# Add logging to all agents
for agent in [coordinator_agent, flight_agent, hotel_agent, activity_agent]:
```

```
    agent.before_agent_callback = log_agent_activity
    agent.before_tool_callback = log_tool_usage

# --- Usage Example ---
async def main():
    # Setup
    session_service = InMemorySessionService()
    runner = Runner(
        agent=coordinator_agent,
        app_name="travel_assistant",
        session_service=session_service
    )

    # Create session
    session = await session_service.create_session(
        app_name="travel_assistant",
        user_id="traveler123",
        session_id="trip_planning_001"
    )

    # Simulate conversation
    queries = [
        "I want to plan a trip from New York to Paris",
        "Find me flights for December 15th",
        "I need a hotel near the city center from Dec 15-20",
        "What activities are available? I'm interested in culture and food"
    ]

    for query in queries:
        print(f"\n--- User: {query} ---")

        user_message = types.Content(
            role='user',
            parts=[types.Part(text=query)]
        )

        async for event in runner.run_async(
            user_id="traveler123",
            session_id="trip_planning_001",
            new_message=user_message
        ):
            if event.is_final_response():
                response = event.content.parts[^0].text
                print(f"Assistant: {response}")
                break
            elif event.is_agent_transfer():
                print(f"→ Transferred to: {event.target_agent}")

if __name__ == "__main__":
    asyncio.run(main())
```

This comprehensive reference manual covers all major ADK components and patterns. Use it as your definitive guide for building sophisticated AI agents with Google's Agent Development Kit. Each section can be expanded with more specific examples based on your particular use cases and requirements.

⁂

1. https://google.github.io/adk-docs/
2. https://google.github.io/adk-docs/agents/multi-agents/
3. https://wandb.ai/google_articles/articles/reports/Google-Agent-Development-Kit-ADK-A-hands-on-tutorial--VmlldzoxMzM2NTIwMQ
4. https://google.github.io/adk-docs/tutorials/
5. https://cloud.google.com/vertex-ai/generative-ai/docs/agent-engine/develop/adk
6. https://www.youtube.com/watch?v=RFFcBkSupxk