ADK Cheat Sheet - Complete Reference

Description: Complete, actionable ADK reference with decision trees, copy-paste patterns, state management, workflows, and production checklists. Everything you need.

ADK Cheat Sheet - Complete Reference

Source: google/adk-python (https://github.com/google/adk-python) (ADK 1.16+)

Last Updated: October 2025

1️⃣ Agent Creation (5 Minutes)

Minimal Agent

```python
from google.adk.agents import Agent

root_agent = Agent(
    name="assistant",
    model="gemini-2.0-flash",
    instruction="You are a helpful assistant."
)
```

Agent with Description

```
root_agent = Agent(
    name="calculator",
    model="gemini-2.0-flash",
    description="Performs mathematical calculations",
    instruction="Use tools to compute calculations accurately."
)
```

Agent with Tools

```
def add_numbers(a: int, b: int) -> dict:
    """Add two numbers together."""
    return {
        "status": "success",
        "result": a + b,
        "report": f"{a} + {b} = {a + b}"
    }

root_agent = Agent(
    name="calculator",
    model="gemini-2.0-flash",
    instruction="Help users with calculations.",
    tools=[add_numbers]
)
```

Agent with Output Key (Auto-save)

```
root_agent = Agent(
    name="analyzer",
    model="gemini-2.0-flash",
    instruction="Analyze the provided data.",
    output_key="analysis_result"  # Saves response to state
)
```

2️⃣ Running Agents

CLI (Web Interface - Recommended for Development)

```
# Start dev UI with agent dropdown
adk web

# Select agent from dropdown UI on http://localhost:8000
```

Programmatic Execution

```python
from google.adk.runners import Runner
from google.genai import types

async def run_agent_example():
    runner = Runner(agent=root_agent)
    session = await runner.session_service.create_session(
        app_name="my_app",
        user_id="user_123"
    )

    new_message = types.Content(
        role="user",
        parts=[types.Part(text="Hello!")]
    )

    async for event in runner.run_async(
        user_id="user_123",
        session_id=session.id,
        new_message=new_message
    ):
        if event.content and event.content.parts:
            print(event.content.parts[0].text)
```

3️⃣ Workflow Decision Tree

Choose the right workflow type:

```
┌─────────────────────────────────────────┐
│   Multiple tasks needed?                │
└────────────────┬────────────────────────┘
                 │
        ┌────────┴────────┐
        │                 │
    ❌ NO           ✓ YES
        │                 │
        │         ┌───────┴────────┐
        │         │                │
        │    Order matters?   No order
        │     │         │      │
        │   YES      NO    YES   NO
        │     │       │     │     │
        ▼     ▼       ▼     ▼     ▼
     Agent  Need      Loop  Para-  Agent
           Iter?   Agent  llel    w/
                          Agent  tools
                          ✓
     ✓     NO ✓
          │
          YES
          │
          ▼
        Loop
        Agent
        ✓
```

4️⃣ Workflow Patterns

Sequential Agent (One After Another)

Use when: Tasks MUST happen in order, each needs previous output

```python
from google.adk.agents import SequentialAgent

research = Agent(
    name="research",
    instruction="Research the topic.",
    output_key="findings"
)

write = Agent(
    name="write",
    instruction="Write article based on: {findings}",
    output_key="article"
)

pipeline = SequentialAgent(
    name="BlogPipeline",
    sub_agents=[research, write],
    description="Research then write blog"
)

root_agent = pipeline
```

Parallel Agent (Simultaneous Execution)

Use when: Tasks are independent, speed matters

```python
from google.adk.agents import ParallelAgent

search_flights = Agent(name="flights", instruction="...")
search_hotels = Agent(name="hotels", instruction="...")
find_activities = Agent(name="activities", instruction="...")

travel_search = ParallelAgent(
    name="TravelSearch",
    sub_agents=[search_flights, search_hotels, find_activities],
    description="Search flights, hotels, activities in parallel"
)

root_agent = travel_search
```

Loop Agent (Iterative Refinement)

Use when: Quality over speed, need iteration (write → critique → improve)

```
from google.adk.agents import LoopAgent

write_draft = Agent(name="writer", instruction="Write essay...")

def exit_loop(tool_context):
    """Call when done."""
    tool_context.actions.end_of_agent = True
    return {"status": "success"}

critic = Agent(
    name="critic",
    instruction="Critique the draft. If perfect say: APPROVE",
    output_key="feedback"
)

improve = Agent(
    name="improver",
    instruction="Improve based on feedback: {feedback}. "
                "If feedback says APPROVE, call exit_loop.",
    tools=[exit_loop],
    output_key="improved_draft"
)

refinement_loop = LoopAgent(
    sub_agents=[critic, improve],
    max_iterations=5
)

root_agent = refinement_loop
```

Fan-Out/Gather (Parallel + Sequential)

Use when: Gather data from multiple sources, then synthesize

```python
from google.adk.agents import ParallelAgent, SequentialAgent

# PARALLEL: Gather from multiple sources
parallel_search = ParallelAgent(
    name="DataGathering",
    sub_agents=[web_searcher, database_lookup, api_query]
)

# SEQUENTIAL: Synthesize results
synthesizer = Agent(
    name="synthesizer",
    instruction="Combine the gathered data into summary"
)

# COMBINE: Parallel gather + Sequential synthesis
fan_out_gather = SequentialAgent(
    name="FanOutGather",
    sub_agents=[parallel_search, synthesizer]
)

root_agent = fan_out_gather
```

5️⃣ Tool Patterns

Function Tool (Most Common)

```python
def search_database(query: str, tool_context) -> dict:
    """
    Search database for information.

    Args:
        query: Search query string

    Returns:
        Dict with status, report, and data
    """
    try:
        results = db.search(query)
        return {
            "status": "success",
            "report": f"Found {len(results)} results",
            "data": results,
            "result_count": len(results)
        }
    except Exception as e:
        return {
            "status": "error",
            "error": str(e),
            "report": f"Search failed: {str(e)}"
        }

agent = Agent(..., tools=[search_database])
```

Tool Return Format (Standard)

```
# ✔️ CORRECT
{
    "status": "success" or "error",      # Required
    "report": "Human-readable message",  # Required
    "data": actual_result,               # Optional
    "count": 42                          # Optional custom fields
}

# ❌ WRONG
{
    "result": "just_the_data",          # Missing status/report
    "error_code": 500                   # Not structured
}
```

OpenAPI Tool (REST APIs)

```python
from google.adk.tools.openapi_toolset import OpenAPIToolset

# From OpenAPI spec
toolset = OpenAPIToolset(
    spec="https://api.example.com/openapi.json",
    auth_config={"type": "bearer", "token": "your-token"}
)

agent = Agent(..., tools=[toolset])
```

MCP Tool (Filesystem, Database)

```python
from google.adk.tools.mcp_toolset import MCPToolset

# Filesystem access
fs_tools = MCPToolset(server="filesystem", path="/allowed/path")

# PostgreSQL database
db_tools = MCPToolset(
    server="postgresql",
    connection_string="postgresql://user:pass@host/db"
)

agent = Agent(..., tools=[fs_tools, db_tools])
```

Built-in Tools

```python
from google.adk.tools.google_search_tool import GoogleSearchTool
from google.adk.tools.code_execution_tool import CodeExecutionTool

agent = Agent(
    ...,
    tools=[
        GoogleSearchTool(),        # Web search
        CodeExecutionTool(),       # Python execution
    ]
)
```

6️⃣ State Management

State Scopes Quick Reference

Scope	Persistence	Use Case	Example
`None` (session)	SessionService dependent	Current task	`state['counter'] = 5`
`user:`	Persistent across sessions	User preferences	`state['user:language'] = 'en'`
`app:`	Global, all users	App settings	`state['app:version'] = '1.0'`
`temp:`	**Never persisted**	Temp calculations	`state['temp:score'] = 85`

Using State in Tools

```python
def save_preference(language: str, tool_context) -> dict:
    # Persistent user preference
    tool_context.state['user:language'] = language

    # Session-level data
    tool_context.state['current_language'] = language

    # Temporary data
    tool_context.state['temp:calculation'] = len(language)

    return {"status": "success", "report": "Preferences saved"}

def get_user_history(tool_context) -> dict:
    # Read user's persistent data
    language = tool_context.state.get('user:language', 'en')
    history = tool_context.state.get('user:history', [])

    return {
        "status": "success",
        "report": f"User language: {language}",
        "data": {"language": language, "history": history}
    }
```

State in Agent Instructions

```python
agent = Agent(
    name="personalized_assistant",
    instruction=(
        "You are helping a user.\n"
        "User's preferred language: {user:language}\n"
        "Current topic: {current_topic}\n"
        "\n"
        "Respond in their preferred language and about the topic."
    )
)
```

State Best Practices

```python
# ✔️ DO: Use appropriate scopes
state['user:preferences'] = {...}       # User-level persistent
state['current_task'] = 'pending'        # Session-level
state['temp:calculation'] = 42           # Temporary only

# ❌ DON'T: Wrong scopes
state['preferences'] = {...}             # Should be user:preferences
state['user:temp_var'] = x               # Should be temp:temp_var

# ✔️ DO: Safe reads with defaults
language = state.get('user:language', 'en')

# ❌ DON'T: Unsafe access
language = state['user:language']  # Fails if not set!

# ✔️ DO: Check before updating
if 'user:history' not in state:
    state['user:history'] = []
state['user:history'].append(item)

# ✔️ DO: Use output_key for auto-save
agent = Agent(..., output_key="task_result")
# Response auto-saved to state['task_result']
```

7️⃣ Common Patterns & Examples

Retry Logic

```python
from typing import Any
import time

def retry_api_call(
    endpoint: str,
    retries: int = 3,
    tool_context = None
) -> dict:
    """Call API with retry logic."""
    for attempt in range(retries):
        try:
            response = requests.get(endpoint, timeout=5)
            response.raise_for_status()
            return {
                "status": "success",
                "report": f"Success on attempt {attempt + 1}",
                "data": response.json()
            }
        except requests.RequestException as e:
            if attempt == retries - 1:
                return {
                    "status": "error",
                    "error": str(e),
                    "report": f"Failed after {retries} attempts"
                }
            time.sleep(2 ** attempt)  # Exponential backoff
    return {"status": "error", "report": "Unknown error"}
```

Caching

```python
from functools import lru_cache
import time

@lru_cache(maxsize=128)
def get_cached_data(key: str) -> str:
    """Cached data lookup."""
    # Expensive operation
    return fetch_from_api(key)

def cache_operation(query: str, tool_context) -> dict:
    """Tool with TTL-based caching."""
    cache_key = f"search:{query}"

    # Check if in cache
    if cache_key in tool_context.state:
        cached_value, timestamp = tool_context.state[cache_key]
        if time.time() - timestamp < 300:  # 5 minute TTL
            return {
                "status": "success",
                "report": "Result from cache",
                "data": cached_value
            }

    # Fresh lookup
    result = search_api(query)
    tool_context.state[cache_key] = (result, time.time())

    return {
        "status": "success",
        "report": "Fresh result",
        "data": result
    }
```

Validation

```python
def validate_input(user_input: str, tool_context) -> dict:
    """Validate and sanitize user input."""
    # Length check
    if not user_input or len(user_input) > 1000:
        return {
            "status": "error",
            "report": "Input must be 1-1000 characters"
        }

    # Check for injection patterns
    dangerous_patterns = ['DROP TABLE', '<script>', '{{', ']]']
    for pattern in dangerous_patterns:
        if pattern.lower() in user_input.lower():
            return {
                "status": "error",
                "report": "Input contains invalid patterns"
            }

    # Sanitize
    clean_input = user_input.strip()

    return {
        "status": "success",
        "report": "Input validated",
        "data": clean_input
    }
```

8️⃣ Environment Setup

Authentication

```
# Google AI Studio (Development)

# Vertex AI (Production)

# Verify
gcloud auth application-default login
```

Model Selection

```
# Fast responses, lower cost
Agent(model="gemini-2.0-flash")

# High quality, reasoning
Agent(model="gemini-2.0-flash-thinking")

# Previous generation
Agent(model="gemini-1.5-flash")
Agent(model="gemini-1.5-pro")

# Other LLMs (if supported)
Agent(model="claude-3.5-sonnet")  # Anthropic
Agent(model="gpt-4-turbo")         # OpenAI
```

9️⃣ Testing & Debugging

Unit Test Template

```
import pytest
from google.adk.agents import Agent
from google.adk.runners import Runner

class TestMyAgent:
    @pytest.fixture
    def agent(self):
        return Agent(
            name="test_agent",
            model="gemini-2.0-flash",
            instruction="Test instruction"
        )

    @pytest.mark.asyncio
    async def test_basic_response(self, agent):
        runner = Runner(agent=agent)
        session = await runner.session_service.create_session(
            app_name="test_app",
            user_id="test_user"
        )

        from google.genai import types
        message = types.Content(
            role="user",
            parts=[types.Part(text="Hello")]
        )

        responses = []
        async for event in runner.run_async(
            user_id="test_user",
            session_id=session.id,
            new_message=message
        ):
            if event.content and event.content.parts:
                responses.append(event.content.parts[0].text)

        assert len(responses) > 0
        assert "hello" in responses[-1].lower()
```

Run Tests

```
# All tests
pytest tests/ -v

# With coverage
pytest tests/ --cov=src --cov-report=html

# Specific test
pytest tests/test_agent.py::TestMyAgent::test_basic_response -v

# Show print statements
pytest tests/ -s
```

Debugging in Web UI

1. Start: `adk web`
2. Open: `http://localhost:8000`
3. Select agent from dropdown
4. Type message
5. Click "Events" tab to see:
6. Agent execution flow
7. Tool calls
8. State changes
9. Errors

🔟 CLI Commands

Development

```
adk web                       # Start dev UI
adk web --debug              # Debug mode
adk web --port 8080          # Custom port
```

Deployment

```
# Cloud Run
adk deploy cloud_run \
  --project my-project \
  --region us-central1 \
  --service-name my-agent

# Vertex AI Agent Engine
adk deploy agent_engine \
  --project my-project \
  --region us-central1

# GKE
adk deploy gke \
  --project my-project \
  --cluster my-cluster
```

1️⃣1️⃣ Production Checklist

Before Deployment

- [ ] All tests passing (100% critical)
- [ ] Error handling for all tool failures
- [ ] Input validation on all tools
- [ ] Rate limiting configured
- [ ] Secrets in Secret Manager (NOT hardcoded)
- [ ] Logging and monitoring setup
- [ ] Performance benchmarks meet SLAs
- [ ] Security review completed
- [ ] Documentation complete

During Deployment

- [ ] Staged rollout (dev → staging → prod)

- [ ] Health checks configured
- [ ] Auto-scaling enabled
- [ ] Alerts configured
- [ ] Rollback plan ready
- [ ] On-call rotation scheduled

After Deployment

- [ ] Monitor error rates
- [ ] Check response times
- [ ] Review logs for issues
- [ ] Measure SLI/SLO compliance
- [ ] Collect user feedback
- [ ] Plan optimizations

1️⃣2️⃣ Best Practices Checklists

Agent Design

- [ ] Single responsibility (one clear purpose)
- [ ] Descriptive name (`content_writer` not `agent1`)
- [ ] Clear, specific instructions (not vague)
- [ ] Error handling with helpful messages
- [ ] Appropriate model for task (balance speed/quality)

Tool Development

- [ ] Returns `{"status", "report", "data"}`
- [ ] Docstring explains what tool does
- [ ] Validates all inputs
- [ ] Handles errors gracefully
- [ ] Idempotent (safe to call multiple times)

State Management

- [ ] Uses correct scope (`user:`, `temp:`, `app:`)
- [ ] Descriptive key names
- [ ] Safe reads with `.get()` and defaults
- [ ] Cleans up old/unused state

Performance

- [ ] Use `ParallelAgent` for independent tasks
- [ ] Cache expensive operations
- [ ] Use streaming for long outputs
- [ ] Choose appropriate model (flash vs pro)
- [ ] Monitor response times

Security

- [ ] Validate all user inputs
- [ ] Sanitize before use
- [ ] Never hardcode secrets
- [ ] Use Secret Manager for production
- [ ] Log security events

1️⃣3️⃣ Quick Troubleshooting

Problem	Solution
Agent doesn't use tool	Check docstring and parameter names
State not persisting	Use persistent SessionService
Slow responses	Use `gemini-2.0-flash`, enable caching
Memory errors	Reduce context, use streaming
Tool not found	Check `adk web` - make sure agent is discoverable
Import errors	Run `pip install -e .` in tutorial dir
Auth fails	Check `GOOGLE_API_KEY` and `GOOGLE_CLOUD_PROJECT`

1️⃣4️⃣ Comparison Tables

Agent Types

Type	Execution	Use Case
`Agent`	Single LLM call	Basic tasks, conversations
`SequentialAgent`	One after another	Pipelines, step-by-step
`ParallelAgent`	All simultaneous	Independent tasks, speed
`LoopAgent`	Repeated until done	Refinement, iteration

Tool Types

Type	Use	Complexity
Function	Python functions	Low
OpenAPI	REST APIs	Medium
MCP	Standard protocol	High
Built-in	Google tools	Low

State Scopes

Scope	Persistence	Speed	Sharing
None	Session-dependent	Fast	Agents in invocation
`user:`	Across sessions	Medium	Per user
`app:`	Global	Slow	All users
`temp:`	Never	Fast	Invocation only

1️⃣5️⃣ Quick Links & Resources

- **Official Docs**: google.github.io/adk-docs (https://google.github.io/adk-docs)
- **GitHub**: github.com/google/adk-python (https://github.com/google/adk-python)
- **API Reference**: google/adk-python API docs (https://github.com/google/adk-python)
- **Tutorials**: Tutorial Index (tutorial_index.md)
- **Mental Models**: Agent Architecture (agent-architecture.md)
- **Glossary**: ADK Glossary (glossary.md)

Version: ADK 1.16+ | **Updated**: October 2025

Pro Tip: Bookmark this page! Use Ctrl/Cmd+F to search for patterns you need.

Generated on 2025-10-21 09:03:12 from adk-cheat-sheet.md

Source: Google ADK Training Hub