Tools & Capabilities

Description: Understanding ADK's tool ecosystem - Function tools, OpenAPI tools, MCP tools, and built-in capabilities

Tools & Capabilities

🎯 **Purpose**: Master ADK's tool ecosystem to extend agent capabilities beyond LLM reasoning.

📚 **Source of Truth**: google/adk-python/src/google/adk/tools/ (https://github.com/google/adk-python/tree/main/src/google/adk/tools/) (ADK 1.15) + Tool implementation patterns

🔧 Tool Ecosystem Overview

Mental Model: Tools are like **power tools** that extend agent capabilities beyond reasoning:

```
┌──────────────────────────────────────────────────────────────┐
│                      TOOL ECOSYSTEM                          │
├──────────────────────────────────────────────────────────────┤
│                                                              │
│ [TOOLS] FUNCTION TOOLS (Custom Skills)                       │
│    "Python functions = agent capabilities"                   │
│    def search_database(query: str) -> dict:                  │
│        return {...}                                          │
│    Use: Custom business logic                                │
│    Source: tools/function_tool.py                            │
│                                                              │
│ [API] OPENAPI TOOLS (API Access)                             │
│    "REST APIs automatically become agent tools"              │
│    OpenAPIToolset(spec_url="https://api.com/spec.json")      │
│    Use: External services, third-party APIs                  │
│    Source: tools/openapi_toolset.py                          │
│                                                              │
│ [MCP] MCP TOOLS (Standardized Protocol)                      │
│    "Model Context Protocol = universal tool language"        │
│    MCPToolset(server="filesystem", path="/data")             │
│    Use: Filesystem, databases, standard services             │
│    Source: tools/mcp_tool/                                   │
│                                                              │
│ [BUILTIN] BUILTIN TOOLS (Google Cloud)                       │
│    "Pre-built Google capabilities"                           │
│    - google_search (web grounding)                           │
│    - google_maps_grounding (location)                        │
│    - Code execution (Python in model)                        │
│    Use: Search, maps, code, enterprise data                  │
│    Source: tools/google_*_tool.py                            │
│                                                              │
│ [FRAMEWORK] FRAMEWORK TOOLS (Third-party)                    │
│    "100+ tools from LangChain/CrewAI"                        │
│    LangchainTool(tool=TavilySearchResults())                 │
│    CrewaiTool(tool=SerperDevTool(), name="search")           │
│    Use: Leverage existing tool ecosystems                    │
│    Source: tools/third_party/                                │
│                                                              │
└──────────────────────────────────────────────────────────────┘
```

🔧 Function Tools (Custom Logic)

Basic Function Tool Pattern

Mental Model: Python functions become callable agent capabilities:

```python
from google.adk.tools import FunctionTool

def search_database(query: str, limit: int = 10) -> Dict[str, Any]:
    """
    Search the company database for relevant information.

    Args:
        query: Search query string
        limit: Maximum number of results to return

    Returns:
        Dict with search results and metadata
    """
    try:
        # Your custom logic here
        results = database.search(query, limit=limit)

        return {
            'status': 'success',
            'report': f'Found {len(results)} results for "{query}"',
            'data': {
                'query': query,
                'results': results,
                'total_found': len(results)
            }
        }
    except Exception as e:
        return {
            'status': 'error',
            'error': str(e),
            'report': f'Database search failed: {str(e)}'
        }

# Create the tool
search_tool = FunctionTool(
    name="search_database",
    description="Search company database for information",
    function=search_database
)

# Use in agent
agent = Agent(
    name="database_assistant",
    model="gemini-2.5-flash",
    tools=[search_tool],
    instruction="Help users search and analyze company data"
)
```

Function Tool Best Practices

Return Format Standard:

```
# Always return structured dict
{
    'status': 'success' | 'error',
    'report': 'Human-readable message',
    'data': { ... }  # Optional structured data
}
```

Error Handling:

```
def robust_tool(param: str) -> Dict[str, Any]:
    try:
        # Main logic
        result = risky_operation(param)
        return {
            'status': 'success',
            'report': f'Successfully processed {param}',
            'data': result
        }
    except ValueError as e:
        return {
            'status': 'error',
            'error': f'Invalid input: {str(e)}',
            'report': f'Could not process {param} due to invalid input'
        }
    except Exception as e:
        return {
            'status': 'error',
            'error': str(e),
            'report': 'An unexpected error occurred'
        }
```

Tool Design Principles:

1. **Single Responsibility**: One tool, one clear purpose
2. **Structured Returns**: Always return the standard format

3. **Comprehensive Error Handling**: Handle all expected error cases
4. **Clear Documentation**: Detailed docstrings with examples
5. **Idempotent**: Safe to call multiple times with same inputs

🌐 OpenAPI Tools (REST API Integration)

Automatic API Tool Generation

Mental Model: REST APIs become agent tools automatically:

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

# Load API specification
api_tools = OpenAPIToolset(
    spec_url="https://api.github.com/swagger.json",
    # or spec_dict=loaded_spec_dict
)

# Tools are automatically created from the API spec
# - get_repos (GET /repos)
# - create_issue (POST /repos/issues)
# - search_code (GET /search/code)
# etc.

# Use in agent
agent = Agent(
    name="github_assistant",
    model="gemini-2.5-flash",
    tools=api_tools.get_tools(),  # Get all generated tools
    instruction="Help users work with GitHub repositories and issues"
)
```

OpenAPI Tool Features

Automatic Parameter Mapping:

```
# API Spec: GET /repos/{owner}/{repo}/issues
# Becomes tool: get_issues(owner: str, repo: str, state?: str)

# Agent can call it naturally:
# "Show me open issues in google/adk repo"
# → Calls get_issues(owner="google", repo="adk", state="open")
```

Authentication Handling:

```
# With API key
api_tools = OpenAPIToolset(
    spec_url="https://api.service.com/spec.json",
    auth_config={
        'type': 'bearer',
        'token': os.getenv('API_TOKEN')
    }
)

# With OAuth2
api_tools = OpenAPIToolset(
    spec_url="https://api.service.com/spec.json",
    auth_config={
        'type': 'oauth2',
        'client_id': '...',
        'client_secret': '...',
        'token_url': 'https://api.service.com/oauth/token'
    }
)
```

Common OpenAPI Patterns

CRUD Operations:

```
# Database API
db_tools = OpenAPIToolset(spec_url="https://db-api.company.com/spec.json")
# Creates: create_record, read_record, update_record, delete_record

# File Storage API
storage_tools = OpenAPIToolset(spec_url="https://storage.company.com/spec.json
# Creates: upload_file, download_file, list_files, delete_file

# Communication API
comm_tools = OpenAPIToolset(spec_url="https://slack.company.com/spec.json")
# Creates: send_message, create_channel, invite_user
```

🔌 MCP Tools (Model Context Protocol)

MCP Architecture

Mental Model: MCP is like **USB for tools** (universal connector):

```
BEFORE MCP (Custom Integrations)
   Agent ──custom──► Filesystem
   Agent ──custom──► Database
   Agent ──custom──► API Service
        (Every integration is different)

AFTER MCP (Standardized Protocol)
   Agent ───MCP────► MCP Server (Filesystem)
   Agent ───MCP────► MCP Server (Database)
   Agent ───MCP────► MCP Server (API Service)
        (One protocol, many servers)
```

MCP Tool Usage

Stdio Connection (Local):

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

# Filesystem access
filesystem_tools = MCPToolset(
    connection_params=StdioConnectionParams(
        command='npx',
        args=['-y', '@modelcontextprotocol/server-filesystem', '/data']
    )
)

# Database access
db_tools = MCPToolset(
    connection_params=StdioConnectionParams(
        command='npx',
        args=['-y', '@modelcontextprotocol/server-sqlite', '--db-path', '/data
    )
)

# Use in agent
agent = Agent(
    name="data_analyst",
    model="gemini-2.5-flash",
    tools=filesystem_tools.get_tools() + db_tools.get_tools(),
    instruction="Analyze data from files and databases"
)
```

HTTP Connection (Remote):

```python
# Remote MCP server
remote_tools = MCPToolset(
    connection_params=HttpConnectionParams(
        url='https://mcp-server.company.com'
    )
)
```

MCP vs Custom Tools

Aspect	Custom Tools	MCP Tools
Setup	Write Python code	Install MCP server
Reusability	Single agent	Any agent
Discovery	Manual	Automatic
Authentication	Custom	Built-in OAuth2
Community	N/A	100+ servers

🏢 Built-in Tools (Google Cloud)

Google Search (Web Grounding)

Mental Model: Connects LLM imagination to real-world facts:

```python
from google.adk.tools import google_search

# Automatic with Gemini 2.0+
agent = Agent(
    name="researcher",
    model="gemini-2.0-flash",  # Built-in search
    instruction="Research topics using web search"
)

# Explicit tool usage
search_agent = Agent(
    name="web_searcher",
    model="gemini-2.5-flash",
    tools=[google_search],
    instruction="Search the web for current information"
)
```

Search Capabilities:

- Real-time web results
- Factual grounding
- Current events and data
- Source citations

Google Maps Grounding

Mental Model: Location intelligence for spatial reasoning:

```
from google.adk.tools import google_maps_grounding

location_agent = Agent(
    name="location_assistant",
    model="gemini-2.5-flash",
    tools=[google_maps_grounding],
    instruction="Help users with location-based queries and directions"
)

# Capabilities:
# - Address resolution
# - Distance calculations
# - Points of interest
# - Route planning
```

Code Execution

Mental Model: Python interpreter built into the model:

```
# Gemini 2.0+ has built-in code execution
code_agent = Agent(
    name="programmer",
    model="gemini-2.0-flash",  # Built-in code execution
    instruction="Write and test Python code"
)

# Can execute code like:
# "Calculate the factorial of 10"
# "Plot a sine wave"
# "Process this CSV data"
```

🔗 Framework Tools (Third-party)

LangChain Integration

Mental Model: Leverage LangChain's 50+ tools:

```
from google.adk.tools.third_party import LangchainTool
from langchain_community.tools import TavilySearchResults

# Wrap LangChain tool
search_tool = LangchainTool(
    tool=TavilySearchResults(max_results=5),
    name="web_search",
    description="Search the web using Tavily"
)

agent = Agent(
    name="research_assistant",
    model="gemini-2.5-flash",
    tools=[search_tool],
    instruction="Research topics using web search"
)
```

CrewAI Integration

Mental Model: Use CrewAI's specialized tools:

```python
from google.adk.tools.third_party import CrewaiTool
from crewai_tools import SerperDevTool

# Wrap CrewAI tool
search_tool = CrewaiTool(
    tool=SerperDevTool(),
    name="google_search",
    description="Search Google using Serper"
)

agent = Agent(
    name="web_researcher",
    model="gemini-2.5-flash",
    tools=[search_tool],
    instruction="Find information using Google search"
)
```

⚡ Parallel Tool Execution

Automatic Parallelization

Mental Model: Multiple tools run simultaneously via `asyncio.gather()`:

```
User: "Check weather in SF, LA, NYC"
         │
    LLM generates 3 FunctionCalls
         │
    ┌────┴────┐
    │ ADK     │  asyncio.gather()
    │ Runtime │  ────────────────────►
    └─────────┘
         │
    ┌────┴────┬────────┬────────┐
    │         │        │        │
 Task A    Task B    Task C   (Parallel)
    SF        LA         NYC
    │         │        │
    └────┬────┴────────┘
         │
    Merge results
         │
    Return to LLM
```

Performance Benefits:

- **Speed**: Independent tasks run in parallel
- **Cost**: Same token cost, faster execution
- **Scalability**: Handle multiple requests simultaneously

Parallel Tool Patterns

Fan-out/Fan-in:

```
# Research multiple sources in parallel
parallel_research = ParallelAgent(
    sub_agents=[
        web_search_agent,
        database_search_agent,
        api_search_agent
    ]
)

# Then merge results
merger_agent = Agent(
    name="result_merger",
    model="gemini-2.5-flash",
    instruction="Combine and summarize research results from multiple sources"
)

# Complete pipeline
research_pipeline = SequentialAgent(
    sub_agents=[parallel_research, merger_agent]
)
```

[TOOLS] Tool Selection Decision Tree

```
Need a Capability?
    |
    ├─ Python Code?
    |   └─ FunctionTool ✓
    |
    ├─ REST API?
    |   └─ OpenAPIToolset ✓
    |
    ├─ Filesystem/DB?
    |   └─ MCPToolset ✓
    |
    ├─ Web/Maps?
    |   └─ Builtin Tools ✓
    |
    └─ Third-party?
        └─ Framework Tools ✓
```

Tool Selection Matrix

Use Case	FunctionTool	OpenAPIToolset	MCPToolset	Built-in	Framework
Custom business logic	✔️	❌	❌	❌	❌
REST API integration	❌	✔️	❌	❌	❌
File system access	❌	❌	✔️	❌	❌
Web search	❌	❌	❌	✔️	✔️
Location services	❌	❌	❌	✔️	❌
Code execution	❌	❌	❌	✔️	❌
Existing tool reuse	❌	❌	❌	❌	✔️

🔧 Tool Development Best Practices

Tool Design Principles

1. **Clear Purpose**: Each tool does one thing well
2. **Consistent Interface**: Standard return format across all tools
3. **Error Resilience**: Graceful failure handling
4. **Performance Aware**: Consider execution time and resource usage
5. **Security Conscious**: Validate inputs, limit access

Tool Testing Patterns

```python
def test_tool():
    # Test success case
    result = search_tool("test query")
    assert result['status'] == 'success'
    assert 'data' in result

    # Test error case
    result = search_tool("")  # Invalid input
    assert result['status'] == 'error'
    assert 'error' in result

    # Test edge cases
    result = search_tool("nonexistent")
    assert result['status'] == 'success'  # Valid query, no results
    assert result['data']['results'] == []
```

Tool Documentation

```python
def comprehensive_tool(
    query: str,
    filters: Optional[Dict[str, Any]] = None,
    limit: int = 100
) -> Dict[str, Any]:
    """
    Comprehensive search across multiple data sources.

    This tool searches databases, APIs, and files to provide
    comprehensive results for user queries.

    Args:
        query: Search query string (required)
        filters: Optional filters to narrow results
            - date_range: {"start": "2024-01-01", "end": "2024-12-31"}
            - categories: ["tech", "business"]
        limit: Maximum results to return (default: 100, max: 1000)

    Returns:
        Dict containing:
        - status: "success" or "error"
        - report: Human-readable summary
        - data: Structured results with metadata

    Examples:
        # Basic search
        tool("machine learning")

        # Filtered search
        tool("AI trends", filters={"categories": ["tech"]}, limit=50)

    Raises:
        No explicit exceptions - all errors returned in result dict
    """
```

🔍 Debugging Tools

Tool Call Inspection

```python
# Enable detailed tool logging
import logging
logging.getLogger('google.adk.tools').setLevel(logging.DEBUG)

# Inspect tool calls in agent responses
result = await runner.run_async(query)
for event in result.events:
    if event.type == 'TOOL_CALL_START':
        print(f"Tool: {event.tool_name}")
        print(f"Args: {event.arguments}")
    elif event.type == 'TOOL_CALL_RESULT':
        print(f"Result: {event.result}")
```

Tool Performance Monitoring

```python
# Track tool execution time
import time

def timed_tool(*args, **kwargs):
    start_time = time.time()
    result = original_tool(*args, **kwargs)
    duration = time.time() - start_time

    # Log performance
    print(f"Tool execution: {duration:.2f}s")

    # Add to result
    result['execution_time'] = duration
    return result
```

📚 Related Topics

- **Agent Architecture → (agent-architecture.md)**: How agents use tools
- **Workflows & Orchestration → (workflows-orchestration.md)**: Coordinating multiple tools
- **LLM Integration → (llm-integration.md)**: How LLMs call tools

🎓 Hands-On Tutorials

- **Tutorial 02: Function Tools (02_function_tools.md)**: Build custom Python function tools
- **Tutorial 03: OpenAPI Tools (03_openapi_tools.md)**: Connect to REST APIs automatically
- **Tutorial 11: Built-in Tools & Grounding (11_built_in_tools_grounding.md)**: Use Google search, maps, and code execution
- **Tutorial 16: MCP Integration (16_mcp_integration.md)**: Standardized tool protocols

🎯 Key Takeaways

1. **Tool Types**: Function for custom logic, OpenAPI for REST APIs, MCP for standards
2. **Built-in Power**: Google tools provide search, maps, code execution
3. **Parallel Execution**: Independent tools run simultaneously for speed
4. **Standard Format**: All tools return `{status, report, data}` structure
5. **Error Handling**: Tools handle errors gracefully, return structured error info

🔗 **Next**: Learn about Workflows & Orchestration (workflows-orchestration.md) to coordinate multiple tools effectively.

Generated on 2025-10-19 17:57:46 from tools-capabilities.md

Source: Google ADK Training Hub