NVIDIA Agent Intelligence Toolkit - Comprehensive Developer Guide

Overview

NVIDIA Agent Intelligence (AIQ) Toolkit is a framework-agnostic, lightweight library for building enterprise AI agents. It treats agents, tools, and workflows as composable function calls, enabling you to build once and reuse across different scenarios.

Key Philosophy: Everything is a function call - agents, tools, and workflows can be composed together in complex applications.

Architecture Patterns

Core Design Principles

1. **Framework Agnostic**: Works alongside LangChain, LlamaIndex, CrewAI, Semantic Kernel
2. **Composability**: Functions can be nested and combined
3. **Declarative Configuration**: YAML-based workflow definitions
4. **Tool Reusability**: Build tools once, use in any agent
5. **MCP Integration**: Full Model Context Protocol support

Agent Types - Deep Dive

1. ReAct Agent (Reasoning + Acting)

Architecture Pattern: Iterative thought-action-observation loop

How It Works

```
User Query → Thought → Action → Tool Call → Observation →
→ (repeat) → Final Answer
```

Key Characteristics:

- Reasons **between** tool calls
- Uses tool names and descriptions for routing
- Iterative decision-making process
- Most flexible but most token-intensive

Configuration

yaml

```yaml
workflow:
  _type: react_agent
  tool_names: [wikipedia_search, current_datetime, code_generation, math_agent]
  llm_name: nim_llm
  verbose: true
  handle_parsing_errors: true
  max_retries: 2
  max_iterations: 15
  max_history: 15
```

As a Nested Function (Agent calling Agent)

yaml

```yaml
functions:
  calculator_multiply:
    _type: calculator_multiply

  calculator_inequality:
    _type: calculator_inequality

  calculator_divide:
    _type: aiq_simple_calculator/calculator_divide

  # Math agent as a tool for parent agent
  math_agent:
    _type: react_agent
    tool_names:
      - calculator_multiply
      - calculator_inequality
      - calculator_divide
    description: 'Useful for performing simple mathematical calculations.'
    llm_name: agent_llm
```

Output Format

The LLM must output in ReAct format:

```
Thought: To answer this question, I need to find information about Dijkstra.
Action: wikipedia_search
Action Input: Dijkstra
Observation: (Wait for tool response...)

Thought: I now know the final answer
Final Answer: Dijkstra was a Dutch computer scientist...
```

Execution Example

Query: "What's the current weather in New York?"

Iteration 1:

- Observation: Question received
- Thought: "I don't have weather data, need to use weather API"
- Action: `weather_api("New York")`

Iteration 2:

- Observation: `72°F, clear skies`
- Thought: "Now I can answer"
- Action: Returns final answer

Configuration Options Explained

Option	Default	Purpose
`tool_names`	Required	List of tools agent can use
`llm_name`	Required	LLM configuration reference
`verbose`	`false`	Enable debug logging
`retry_parsing_errors`	`true`	Auto-retry on format errors
`max_retries`	`1`	Max retry attempts
`max_iterations`	`15`	Max tool calls per query
`max_history`	`15`	Conversation history size
`use_openai_api`	`false`	Output OpenAI API spec
`include_tool_input_schema_in_tool_description`	`true`	Add schemas to descriptions
`system_prompt`	Default	Override system prompt

Custom Prompt Template

```python
```

```
CUSTOM_PROMPT = """
Answer questions using these tools:
{tools}

Format for tool use:
Question: [input question]
Thought: [reasoning]
Action: [tool name from {tool_names}]
Action Input: [tool input or None]
Observation: [wait for response]

Format for final answer:
Thought: I now know the final answer
Final Answer: [answer]
"""
```

Limitations

- **High Token Usage**: Multiple LLM calls per task
- **Prompt Sensitivity**: Requires careful tuning
- **Hallucination Risk**: May output incorrect format
- **Sequential Only**: No parallel execution
- **Complexity in Long Chains**: Error propagation

Best Use Cases

- Complex reasoning tasks requiring adaptation
- Workflows where intermediate reasoning is valuable
- Tasks where you can't predict tool call sequence

- Debug-friendly scenarios (visible reasoning)

---

2. Reasoning Agent

Architecture Pattern: Plan-first execution with reasoning LLM

How It Works

```
User Query → Reasoning (Plan Creation) →
→ Augmented Function Execution → Final Answer
```

Key Characteristics:

- Reasons **before** execution (upfront planning)
- Wraps another agent/function with reasoning layer
- Requires reasoning-capable LLM (DeepSeek R1, etc.)
- Single planning phase, no inter-step reasoning

Configuration

```yaml
workflow:
  _type: reasoning_agent
  llm_name: deepseek_r1_model
  augmented_fn: react_agent  # The agent to reason on top of
  verbose: true
```

Architecture Pattern

```yaml
llms:
  deepseek_r1_model:
    _type: nim_llm
    model_name: deepseek-ai/DeepSeek-R1

functions:
  wikipedia_search:
    _type: wikipedia_search

  calculator:
    _type: calculator

  # Base agent that will be augmented
  base_react_agent:
    _type: react_agent
    tool_names: [wikipedia_search, calculator]
    llm_name: base_llm

# Reasoning agent wrapping the base agent
workflow:
  _type: reasoning_agent
  llm_name: deepseek_r1_model
  augmented_fn: base_react_agent
```

Execution Flow

Step 1: Reasoning Phase

```
Input: "Calculate 15% tip on $45.50 meal"

Reasoning LLM Output:
<think>
The user wants to calculate a tip. I need to:
1. Calculate 15% of $45.50
2. Use the calculator tool
3. Return formatted result
</think>

Plan:
1. Call calculator with input "45.50 * 0.15"
2. Format result as currency
3. Return final answer
```

Step 2: Execution Phase

- Passes plan to augmented function
- Function executes with plan as guidance

Step 3: Response

- Final answer based on execution results

Configuration Options

Option	Default	Purpose
`llm_name`	Required	Reasoning-capable LLM
`augmented_fn`	Required	Agent/function to augment
`verbose`	`false`	Debug logging
`reasoning_prompt_template`	Default	Planning prompt
`instruction_prompt_template`	Default	Execution prompt

Custom Prompts

Reasoning Prompt:

yaml

```yaml
reasoning_prompt_template: |
  You are an expert reasoning model tasked with creating a detailed execution plan
  for a system that has the following description:

  **Description:**
  {augmented_function_desc}

  Given the following input and tools, provide a step-by-step plan:

  **Input:**
  {input_text}

  **Tools:**
  {tools}

  **PLAN:**
```

Instruction Prompt:

```yaml
instruction_prompt_template: |
  Answer the following question based on message history: {input_text}

  Here is a plan for execution:
  {reasoning_output}

  You must respond with the answer to the original question directly.
```

Comparison: With vs Without Reasoning

Without Reasoning Agent:

```
Query → ReAct (iterate) → Answer
- Multiple LLM calls
- Reasons between steps
- Adapts during execution
```

With Reasoning Agent:

```
Query → Reason (plan) → ReAct (execute plan) → Answer
- Upfront comprehensive planning
- Better quality plans
- More efficient tool usage
```

Limitations

- **Requires Reasoning LLM**: Must support `<think>` tags

- **No Dynamic Adaptation**: Can't revise plan during execution
- **Planning Overhead**: Upfront cost for simple tasks
- **LLM Dependency**: Quality depends on reasoning model

Best Use Cases

- Complex multi-step workflows
- Tasks benefiting from comprehensive planning
- When you have access to reasoning models
- Reducing trial-and-error tool calls

---

3. ReWOO Agent (Reasoning Without Observation)

Architecture Pattern: Complete plan-first, then execute

How It Works

```
User Query → Complete Plan (with placeholders) →
→ Execute All Steps → Solve with Evidence → Answer
```

Key Characteristics:

- **Total separation** of planning and execution
- Uses evidence placeholders (`#E1`, `#E2`)
- Most token-efficient
- All tool calls planned upfront

Configuration

```yaml
workflow:
  _type: rewoo_agent
  tool_names: [wikipedia_search, current_datetime, code_generation]
  llm_name: nim_llm
  verbose: true
  use_tool_schema: true
```


Architecture Deep Dive

Phase 1: Planning

```json
```

```json
[
  {
    "plan": "Get today's date",
    "evidence": {
      "placeholder": "#E1",
      "tool": "current_datetime",
      "tool_input": {}
    }
  },
  {
    "plan": "Search for historical weather data",
    "evidence": {
      "placeholder": "#E2",
      "tool": "weather_search",
      "tool_input": "New York weather on #E1 last year"
    }
  },
  {
    "plan": "Compare temperatures",
    "evidence": {
      "placeholder": "#E3",
      "tool": "calculator",
      "tool_input": "Compare #E2 current vs last year"
    }
  }
]
```

Phase 2: Execution

- Execute step 1 → Replace `#E1` with actual date
- Execute step 2 → Use `#E1` value, get result for `#E2`

- Execute step 3 → Use #E2 value, get result for #E3

Phase 3: Solution

- Combine all evidence (#E1, #E2, #E3)
- Generate final natural language answer

Example Walkthrough

Query: "What was the weather in New York last year on this date?"

1. Planning Phase:

python

```
plan = [
  {
    "plan": "Get today's date",
    "evidence": {
      "placeholder": "#E1",
      "tool": "current_datetime",
      "tool_input": {}
    }
  },
  {
    "plan": "Calculate last year's date",
    "evidence": {
      "placeholder": "#E2",
      "tool": "date_calculator",
      "tool_input": "#E1 minus 1 year"
    }
  },
  {
    "plan": "Search historical weather",
    "evidence": {
      "placeholder": "#E3",
      "tool": "weather_api",
      "tool_input": "New York #E2"
    }
  }
]
```

2. Execution Phase:

```
python
```

```
# Step 1
#E1 = current_datetime() → "2025-10-04"

# Step 2 (using #E1)
#E2 = date_calculator("2025-10-04 minus 1 year") → "2024-10-04"

# Step 3 (using #E2)
#E3 = weather_api("New York 2024-10-04") → "68°F, partly cloudy"
```

3. Solution Phase:

```
Evidence collected:
- #E1: 2025-10-04
- #E2: 2024-10-04
- #E3: 68°F, partly cloudy

Final Answer: "The weather in New York on October 4th, 2024
was 68°F with partly cloudy skies."
```

Configuration Options

Option	Default	Purpose
`tool_names`	Required	Available tools
`llm_name`	Required	LLM for planning/solving
`verbose`	`false`	Debug logging
`use_tool_schema`	`true`	Include tool schemas
`include_tool_input_schema_in_tool_description`	`true`	Schema in descriptions
`max_history`	`15`	Conversation history
`use_openai_api`	`false`	API format
`planner_prompt`	Default	Planning prompt
`solver_prompt`	Default	Solution prompt
`additional_instructions`	`None`	Extra instructions

Token Efficiency Comparison

ReAct Agent (Iterative):

```
Prompt 1: [System + Query + Tools] = 1000 tokens
Response 1: [Thought + Action] = 100 tokens
Prompt 2: [System + History + Tools + Observation] = 1300 tokens
Response 2: [Thought + Action] = 100 tokens
...
Total: ~3000+ tokens
```

ReWOO Agent (Decoupled):

```
Prompt 1 (Plan): [System + Query + Tools] = 1000 tokens
Response 1: [Complete Plan with placeholders] = 300 tokens
Execution: [Replace placeholders] = minimal
Prompt 2 (Solve): [Plan + Evidence] = 800 tokens
Response 2: [Final Answer] = 100 tokens
Total: ~2200 tokens (25-30% reduction)
```

Limitations

- **Sequential Execution**: Can't parallelize independent steps
- **Planning Overhead**: Upfront cost for simple queries
- **Limited Adaptability**: Can't revise plan based on results
- **Complex Planning**: Requires good tool understanding
- **Memory Constraints**: Must hold entire plan + evidence

Best Use Cases

- Token-constrained environments
- Predictable multi-step workflows
- When tool outcomes are reliable
- Cost optimization scenarios
- Clear sequential dependencies

---

4. Tool Calling Agent

Architecture Pattern: Direct function invocation via LLM tool calling

How It Works

```
User Query → Function Matching (via schema) →
→ Direct Tool Execution → Response
```

Key Characteristics:

- **No reasoning between calls**
- Uses native LLM tool calling capability
- Relies on tool schemas for routing
- Most efficient for structured tasks
- Requires tool-calling compatible LLM

Configuration

```yaml
workflow:
  _type: tool_calling_agent
  tool_names: [wikipedia_search, current_datetime, code_generation]
  llm_name: nim_llm
  verbose: true
  handle_tool_errors: true
  max_iterations: 15
```

Requirements

```bash
# Install with LangChain support
uv pip install -e '.[langchain]'
```

Architecture Pattern

yaml

```yaml
llms:
  tool_calling_llm:
    _type: nim_llm
    model_name: gpt-4  # Must support tool calling
    supports_tool_calling: true

functions:
  # Tool with explicit schema
  weather_tool:
    _type: weather_api
    input_schema:
      type: object
      properties:
        location:
          type: string
          description: "City name"
        units:
          type: string
          enum: ["celsius", "fahrenheit"]
      required: ["location"]

  # Another tool
  calculator_tool:
    _type: calculator
    input_schema:
      type: object
      properties:
        expression:
          type: string
          description: "Math expression to evaluate"
      required: ["expression"]
```

```yaml
workflow:
  _type: tool_calling_agent
  tool_names: [weather_tool, calculator_tool]
  llm_name: tool_calling_llm
  handle_tool_errors: true
```

Execution Flow

Query: "What's the weather in New York?"

1. Function Matching:

json

```json
{
 "tool_calls": [
   {
    "id": "call_123",
    "type": "function",
    "function": {
     "name": "weather_tool",
     "arguments": "{\"location\": \"New York\", \"units\": \"fahrenheit\"}"
    }
   }
 ]
}
```

2. Direct Execution:

python

```
result = weather_tool(location="New York", units="fahrenheit")
# → "72°F, clear skies"
```

3. Response:

```
"The weather in New York is currently 72°F with clear skies."
```

No intermediate reasoning steps!

Tool Schema Definition

Pydantic Model:

```python
```

```python
from pydantic import BaseModel, Field

class WeatherInput(BaseModel):
    location: str = Field(description="City name or coordinates")
    units: str = Field(
        default="celsius",
        description="Temperature units",
        enum=["celsius", "fahrenheit"]
    )

class WeatherTool:
    name = "weather_api"
    description = "Get current weather for a location"
    input_schema = WeatherInput

    def __call__(self, location: str, units: str = "celsius"):
        # Implementation
        return f"Weather in {location}"
```

JSON Schema:

```
json
```

```json
{
 "name": "weather_api",
 "description": "Get current weather for a location",
 "parameters": {
  "type": "object",
  "properties": {
   "location": {
    "type": "string",
    "description": "City name or coordinates"
   },
   "units": {
    "type": "string",
    "description": "Temperature units",
    "enum": ["celsius", "fahrenheit"]
   }
  },
  "required": ["location"]
 }
}
```

Configuration Options

Option	Default	Purpose
`tool_names`	Required	Available tools
`llm_name`	Required	Tool-calling capable LLM
`verbose`	`false`	Debug logging
`handle_tool_errors`	`true`	Catch and retry on errors
`max_iterations`	`15`	Max tool calls
`description`	Default	Tool description

Error Handling

With `handle_tool_errors: true`:

```python
try:
    result = tool(**args)
except Exception as e:
    # Return error as ToolMessage
    return ToolMessage(
        content=f"Error: {str(e)}",
        tool_call_id=call_id
    )
    # LLM can try again with corrected input
```

Without error handling:

```python
# Error propagates, workflow stops
```

Multi-Tool Calling

Some LLMs support parallel tool calling:

```json
```

```json
{
 "tool_calls": [
  {
   "id": "call_1",
   "function": {
    "name": "weather_tool",
    "arguments": "{\"location\": \"New York\"}"
   }
  },
  {
   "id": "call_2",
   "function": {
    "name": "weather_tool",
    "arguments": "{\"location\": \"London\"}"
   }
  }
 ]
}
```


Both execute simultaneously, results combined.

Comparison with ReAct

Aspect	Tool Calling	ReAct
Reasoning	None	Between every step
Token Usage	Low	High
Flexibility	Low	High
Speed	Fast	Slow
Debugging	Harder	Easier (visible thoughts)
LLM Requirement	Tool calling support	Any LLM
Use Case	Structured tasks	Complex reasoning

Limitations

- **Requires Tool Calling LLM**: GPT-4, Claude, Gemini, etc.
- **No Reasoning**: Can't adapt strategy
- **Schema Dependent**: Needs well-defined schemas
- **Less Flexible**: Can't handle unexpected scenarios
- **Poor Tool Names = Poor Performance**

Best Use Cases

- API orchestration
- Database queries
- Structured data retrieval
- High-throughput scenarios
- Cost-sensitive applications
- Clear tool-task mapping

Agent Comparison Matrix

Feature	ReAct	Reasoning	ReWOO	Tool Calling
Token Efficiency	⭐⭐	⭐⭐⭐	⭐⭐⭐⭐⭐	⭐⭐⭐⭐
Speed	⭐⭐	⭐⭐⭐	⭐⭐⭐⭐	⭐⭐⭐⭐⭐
Flexibility	⭐⭐⭐⭐⭐	⭐⭐⭐⭐	⭐⭐	⭐⭐
Debugging	⭐⭐⭐⭐⭐	⭐⭐⭐⭐	⭐⭐⭐	⭐⭐
Complex Reasoning	⭐⭐⭐⭐	⭐⭐⭐⭐⭐	⭐⭐⭐	⭐
Setup Complexity	⭐⭐⭐	⭐⭐⭐⭐	⭐⭐⭐	⭐⭐
LLM Requirements	Any	Reasoning LLM	Any	Tool Calling

Model Context Protocol (MCP) Integration

MCP Overview

MCP is Anthropic's open protocol for standardizing how applications provide context to LLMs. AIQ Toolkit has **full bidirectional MCP support**.

AIQ as MCP Client

Use Case: Connect to remote MCP servers and use their tools in your workflows

Architecture

```
AIQ Workflow → MCP Client → Remote MCP Server → External Tools
```

Configuration Pattern

yaml

```yaml
functions:
  # Wrap individual MCP tools
  mcp_time_tool:
    _type: mcp_tool_wrapper
    url: "http://localhost:8080/sse"
    mcp_tool_name: get_current_time
    description: "Get current time from MCP server"

  mcp_weather_tool:
    _type: mcp_tool_wrapper
    url: "http://localhost:8080/sse"
    mcp_tool_name: get_weather
    description: "Get weather from MCP server"

  # Local tools
  local_calculator:
    _type: calculator

# Use in agent
workflow:
  _type: react_agent
  tool_names: [mcp_time_tool, mcp_weather_tool, local_calculator]
  llm_name: nim_llm
```

MCP Tool Wrapper

yaml

```yaml
mcp_tool_config:
  _type: mcp_tool_wrapper

  # Required: MCP server URL
  url: "http://localhost:8080/sse"

  # Required: Specific tool name from server
  mcp_tool_name: "tool_name"

  # Optional: Override server description
  description: "Custom description for better routing"
```

Complete Example

1. Start MCP Server (external service):

```bash
# Check if MCP server is running
docker ps --filter "name=mcp-proxy-aiq-time"

# Should show:
# CONTAINER ID   IMAGE            PORTS                  NAMES
# 4279653533ec   time_service     0.0.0.0:8080->8080/tcp   mcp-proxy-aiq-time
```

2. Configure AIQ Workflow:

`config-mcp-date.yml`:

```yaml
```

```yaml
llms:
  nim_llm:
    _type: nim_llm
    model_name: gpt-4

functions:
  # MCP tool
  mcp_time_tool:
    _type: mcp_tool_wrapper
    url: "http://localhost:8080/sse"
    mcp_tool_name: get_current_time
    description: "Returns current date/time from MCP server"

  # Local tools
  calculator_multiply:
    _type: calculator_multiply

  calculator_inequality:
    _type: calculator_inequality

workflow:
  _type: react_agent
  tool_names:
    - mcp_time_tool
    - calculator_multiply
    - calculator_inequality
  llm_name: nim_llm
```

3. Run Workflow:

```
bash
```

```
aiq run \
  --config_file config-mcp-date.yml \
  --input "Is the product of 2 * 4 greater than the current hour?"
```

Execution:

```
1. Agent calls mcp_time_tool → MCP Server returns "14:00"
2. Agent calls calculator_multiply(2, 4) → Returns 8
3. Agent calls calculator_inequality(8, 14) → Returns false
4. Agent returns: "No, 2 * 4 (8) is not greater than the current hour (14)"
```

Input Schema Generation

MCP tools automatically generate Pydantic schemas:

MCP Server Schema:

```
json
```

```
{
 "name": "get_current_time",
 "description": "Get current time in specific timezone",
 "inputSchema": {
  "type": "object",
  "properties": {
   "timezone": {
    "type": "string",
    "description": "IANA timezone (e.g., 'America/New_York')"
   }
  },
  "required": ["timezone"]
 }
}
```

Generated Pydantic Model:

```python
class GetCurrentTimeInput(BaseModel):
    timezone: str = Field(description="IANA timezone")
```

Usage in Agent:

```python
# Agent can call with:
mcp_time_tool(timezone="America/New_York")
mcp_time_tool('{"timezone": "America/New_York"}')  # JSON string
mcp_time_tool({"timezone": "America/New_York"})     # Dict
```

Discovering MCP Tools

```bash
# List all tools from MCP server
aiq info mcp --url http://localhost:8080/sse

# Output:
# get_current_time
# convert_time
# get_timezone_info

# Get detailed info about specific tool
aiq info mcp --url http://localhost:8080/sse --tool get_current_time

# Output:
# Tool: get_current_time
# Description: Get current time in a specific timezone
# Input Schema:
# {
#   "properties": {
#     "timezone": {
#       "description": "IANA timezone name",
#       "type": "string"
#     }
#   },
#   "required": ["timezone"]
# }
```

AIQ as MCP Server

Use Case: Expose your AIQ tools as MCP tools for other clients

Architecture

```
AIQ Workflow Tools → MCP Server → MCP Clients (Claude Desktop, etc.)
```

Basic Server

```bash
# Start MCP server exposing all tools from workflow
aiq mcp --config_file examples/simple_calculator/configs/config.yml

# Server starts on http://localhost:9901/sse
```

Filtered Publishing

```bash
# Publish only specific tools
aiq mcp \
  --config_file config.yml \
  --tool_names calculator_multiply \
  --tool_names calculator_divide \
  --tool_names calculator_subtract
```

Complete Example

1. Define Workflow with Tools:

`config.yml`:

```yaml
functions:
  calculator_add:
    _type: calculator_add

  calculator_multiply:
    _type: calculator_multiply

  calculator_divide:
    _type: calculator_divide

  wikipedia_search:
    _type: wikipedia_search

workflow:
  _type: react_agent
  tool_names:
    - calculator_add
    - calculator_multiply
    - calculator_divide
    - wikipedia_search
  llm_name: nim_llm
```

2. Start MCP Server:

```bash
```

```
# Terminal 1: Start server
aiq mcp --config_file config.yml

# Output:
# MCP Server started on http://localhost:9901
# Publishing 4 tools: calculator_add, calculator_multiply, calculator_divide, wikipedia_search
```

3. Verify Published Tools:

```
bash
```

```
# Terminal 2: Check published tools
aiq info mcp

# Output:
# calculator_add
# calculator_multiply
# calculator_divide
# wikipedia_search

# Get detailed schema
aiq info mcp --tool calculator_multiply

# Output:
# Tool: calculator_multiply
# Description: Multiply two numbers together
# Input Schema:
# {
#   "properties": {
#     "text": {
#       "type": "string",
#       "title": "Text"
#     }
#   },
#   "required": ["text"]
# }
```

4. Use from Another AIQ Workflow (as MCP Client):

`config-mcp-client.yml`:

```
yaml
```

```yaml
functions:
  # Remote MCP tools
  remote_multiply:
    _type: mcp_tool_wrapper
    url: "http://localhost:9901/sse"
    mcp_tool_name: calculator_multiply

  remote_divide:
    _type: mcp_tool_wrapper
    url: "http://localhost:9901/sse"
    mcp_tool_name: calculator_divide

  # Local tool
  local_subtract:
    _type: calculator_subtract

workflow:
  _type: tool_calling_agent
  tool_names: [remote_multiply, remote_divide, local_subtract]
  llm_name: nim_llm
```

5. Execute:

```bash
bash
```

```
aiq run \
  --config_file config-mcp-client.yml \
  --input "What is (10 * 5) / 2 minus 3?"

# Flow:
# 1. Calls remote_multiply(10, 5) via MCP → 50
# 2. Calls remote_divide(50, 2) via MCP → 25
# 3. Calls local_subtract(25, 3) → 22
# Result: 22
```

MCP Server Configuration

Custom Host/Port:

```bash
aiq mcp \
  --config_file config.yml \
  --host 0.0.0.0 \
  --port 8080
```

With Environment Variables:

```bash
export AIQ_MCP_HOST=0.0.0.0
export AIQ_MCP_PORT=8080

aiq mcp --config_file config.yml
```

Integration with External MCP Clients

AIQ MCP server is compatible with any MCP client:

Claude Desktop Integration:

json

```json
// claude_desktop_config.json
{
  "mcpServers": {
    "aiq-calculator": {
      "url": "http://localhost:9901/sse"
    }
  }
}
```

Custom Python MCP Client:

python

```python
import httpx
from mcp import ClientSession, StdioServerParameters
from mcp.client.stdio import stdio_client

async def use_aiq_tools():
    async with httpx.AsyncClient() as client:
        # Connect to AIQ MCP server
        response = await client.get(
            "http://localhost:9901/sse/tools/list"
        )
        tools = response.json()

        # Call tool
        result = await client.post(
            "http://localhost:9901/sse/tools/call",
            json={
                "name": "calculator_multiply",
                "arguments": {"text": "5 * 3"}
            }
        )
        print(result.json())
```

Advanced Patterns

1. Hierarchical Agent Architecture

Pattern: Specialized sub-agents coordinated by parent agent

```yaml
yaml
```

```yaml
llms:
  main_llm:
    _type: nim_llm
    model_name: gpt-4

  fast_llm:
    _type: nim_llm
    model_name: gpt-3.5-turbo

functions:
  # Calculation sub-agent
  math_tools:
    calculator_add:
      _type: calculator_add
    calculator_multiply:
      _type: calculator_multiply

  math_agent:
    _type: tool_calling_agent
    tool_names: [calculator_add, calculator_multiply]
    llm_name: fast_llm
    description: "Performs mathematical calculations efficiently"

  # Research sub-agent
  research_tools:
    wikipedia_search:
      _type: wikipedia_search
    web_search:
      _type: web_search

  research_agent:
    _type: react_agent
```

```yaml
    tool_names: [wikipedia_search, web_search]
    llm_name: main_llm
    description: "Researches information from various sources"

  # Code generation sub-agent
  code_agent:
    _type: reasoning_agent
    llm_name: main_llm
    augmented_fn: code_generator
    description: "Generates and debugs code"

# Coordinator agent
workflow:
  _type: react_agent
  tool_names: [math_agent, research_agent, code_agent]
  llm_name: main_llm
  verbose: true
```

Execution:

```
User: "Research Python's creator and calculate their age if born in 1956"

Coordinator Agent:
  Thought: Need to research AND calculate
  Action: research_agent
  Input: "Python programming language creator"

Research Agent:
  → wikipedia_search("Python creator")
  → Returns: "Guido van Rossum, born 1956"

Coordinator Agent:
  Thought: Now calculate age
  Action: math_agent
  Input: "2025 - 1956"

Math Agent:
  → calculator_subtract(2025, 1956)
  → Returns: 69

Coordinator Agent:
  Final Answer: "Guido van Rossum, born 1956, is 69 years old"
```

---

2. Hybrid Agent Pattern

Pattern: Combine different agent types for optimal performance

```yaml
```

```yaml
functions:
  # Fast tool calling for simple operations
  quick_tools_agent:
    _type: tool_calling_agent
    tool_names: [calculator, datetime, string_format]
    llm_name: fast_llm
    description: "Fast execution of simple operations"

  # ReAct for complex reasoning
  reasoning_tools_agent:
    _type: react_agent
    tool_names: [database_query, api_call, data_analysis]
    llm_name: smart_llm
    description: "Complex reasoning tasks"

  # ReWOO for token-efficient multi-step
  pipeline_agent:
    _type: rewoo_agent
    tool_names: [extract, transform, load]
    llm_name: efficient_llm
    description: "Data pipeline operations"

# Router selects appropriate agent
workflow:
  _type: react_agent
  tool_names: [quick_tools_agent, reasoning_tools_agent, pipeline_agent]
  llm_name: router_llm
  system_prompt: |
    Route tasks to agents:
    - quick_tools_agent: Simple, fast operations
```

```
- reasoning_tools_agent: Complex decision making
- pipeline_agent: Multi-step data workflows
```

---

3. MCP Microservices Architecture

Pattern: Distribute tools across multiple MCP servers

Service 1: Math Service

```yaml
# math-service/config.yml
functions:
  calculator_add:
    _type: calculator_add
  calculator_multiply:
    _type: calculator_multiply
  calculator_divide:
    _type: calculator_divide

workflow:
  _type: tool_calling_agent
  tool_names: [calculator_add, calculator_multiply, calculator_divide]
  llm_name: local_llm
```

```bash
# Start on port 9901
aiq mcp --config_file math-service/config.yml --port 9901
```

Service 2: Data Service

```yaml
# data-service/config.yml
functions:
  database_query:
    _type: database_query
  csv_reader:
    _type: csv_reader
  json_parser:
    _type: json_parser

workflow:
  _type: react_agent
  tool_names: [database_query, csv_reader, json_parser]
  llm_name: local_llm
```

```bash
# Start on port 9902
aiq mcp --config_file data-service/config.yml --port 9902
```

Service 3: AI Service

```yaml
```

```yaml
# ai-service/config.yml
functions:
  image_analyzer:
    _type: image_analyzer
  text_summarizer:
    _type: text_summarizer
  sentiment_analyzer:
    _type: sentiment_analyzer

workflow:
  _type: reasoning_agent
  augmented_fn: image_processor
  llm_name: vision_llm
```

```bash
# Start on port 9903
aiq mcp --config_file ai-service/config.yml --port 9903
```

Orchestrator Service

```yaml
```

```yaml
# orchestrator/config.yml
functions:
 # Math service tools
 math_add:
  _type: mcp_tool_wrapper
  url: "http://localhost:9901/sse"
  mcp_tool_name: calculator_add

 math_multiply:
  _type: mcp_tool_wrapper
  url: "http://localhost:9901/sse"
  mcp_tool_name: calculator_multiply

 # Data service tools
 db_query:
  _type: mcp_tool_wrapper
  url: "http://localhost:9902/sse"
  mcp_tool_name: database_query

 csv_read:
  _type: mcp_tool_wrapper
  url: "http://localhost:9902/sse"
  mcp_tool_name: csv_reader

 # AI service tools
 analyze_image:
  _type: mcp_tool_wrapper
  url: "http://localhost:9903/sse"
  mcp_tool_name: image_analyzer

 summarize_text:
  _type: mcp_tool_wrapper
```

```yaml
    url: "http://localhost:9903/sse"
    mcp_tool_name: text_summarizer

workflow:
  _type: react_agent
  tool_names:
    - math_add
    - math_multiply
    - db_query
    - csv_read
    - analyze_image
    - summarize_text
  llm_name: orchestrator_llm
```

```bash
# Run orchestrator
aiq run --config_file orchestrator/config.yml --input "Query sales data and calculate total"
```

Architecture Benefits:

- **Scalability**: Scale individual services independently
- **Isolation**: Service failures don't crash entire system
- **Deployment**: Update services without redeploying everything
- **Load Balancing**: Route requests to multiple instances

Production Patterns

1. Error Handling & Resilience

yaml

```yaml
workflow:
  _type: react_agent
  tool_names: [flaky_api, backup_api]
  llm_name: nim_llm

  # Retry configuration
  max_retries: 3
  retry_parsing_errors: true
  handle_parsing_errors: true

  # Tool error handling
  handle_tool_errors: true

  # Timeout configuration
  max_iterations: 15

  # Logging
  verbose: true
```

Custom Error Handler:

python

```python
from aiqtoolkit import Function

class ResilientAPITool(Function):
    def __call__(self, input_data):
        max_attempts = 3
        for attempt in range(max_attempts):
            try:
                result = self.api_call(input_data)
                return result
            except TimeoutError:
                if attempt == max_attempts - 1:
                    return "Service unavailable, using cached data"
                time.sleep(2 ** attempt)  # Exponential backoff
            except ValueError as e:
                return f"Invalid input: {str(e)}"
```

---

2. Observability & Profiling

```yaml
```

```yaml
workflow:
  _type: react_agent
  tool_names: [tool1, tool2, tool3]
  llm_name: nim_llm
  verbose: true

  # Profiling
  enable_profiling: true

  # OpenTelemetry observability
  enable_tracing: true
  tracing_backend: phoenix  # or weave
```

Profiling Output:

```
Workflow Execution Profile:
├── Agent: react_agent (5.2s)
│   ├── LLM Call 1 (1.2s) - 450 tokens
│   ├── Tool: wikipedia_search (2.1s)
│   ├── LLM Call 2 (0.9s) - 320 tokens
│   └── Tool: calculator (0.05s)
└── Total: 5.2s, 770 tokens
```

---

3. Caching Strategy

```
yaml
```

```yaml
llms:
  cached_llm:
    _type: nim_llm
    model_name: gpt-4
    enable_caching: true
    cache_ttl: 3600  # 1 hour

functions:
  expensive_search:
    _type: wikipedia_search
    enable_caching: true
    cache_backend: redis
    cache_ttl: 1800
```

---

4. Rate Limiting

yaml

```yaml
functions:
  rate_limited_api:
    _type: external_api
    rate_limit:
      requests_per_minute: 60
      burst_size: 10

  expensive_llm_call:
    _type: nim_llm
    rate_limit:
      tokens_per_minute: 10000
      requests_per_minute: 20
```

Development Workflow

1. Local Development

```bash
# Install in development mode
git clone https://github.com/NVIDIA/AIQToolkit.git
cd AIQToolkit
uv pip install -e '.[dev]'

# Install with specific integrations
uv pip install -e '.[langchain,llamaindex]'
```

2. Testing Tools

```yaml
```

```
# test-config.yml
functions:
  test_calculator:
    _type: calculator_add

  test_mock_api:
    _type: mock_api
    responses:
      - input: "test"
        output: "success"

workflow:
  _type: tool_calling_agent
  tool_names: [test_calculator, test_mock_api]
  llm_name: test_llm
```

```bash
# Run tests
aiq run --config_file test-config.yml --input "test query"
```

3. Interactive UI Development

```bash
```

```
# Launch UI for debugging
aiq ui --config_file config.yml

# Opens browser at http://localhost:8000
# Features:
# - Interactive chat
# - Tool call visualization
# - Step-by-step debugging
# - Token usage tracking
```

4. Evaluation

```yaml
# eval-config.yml
evaluation:
  dataset: test-queries.jsonl
  metrics:
    - accuracy
    - latency
    - token_usage
    - tool_call_correctness

  test_cases:
    - input: "What is 2 + 2?"
      expected: "4"
    - input: "Who created Python?"
      expected: "Guido van Rossum"
```

```bash
```

```
aiq evaluate --config_file eval-config.yml
```

Custom Tool Development

Basic Tool

```
python
```

```python
from aiqtoolkit import Function, FunctionBaseConfig
from pydantic import BaseModel, Field

# Input schema
class MyToolInput(BaseModel):
    query: str = Field(description="Query parameter")
    limit: int = Field(default=10, description="Result limit")

# Configuration
class MyToolConfig(FunctionBaseConfig, name="my_custom_tool"):
    api_key: str = Field(description="API key for service")
    endpoint: str = Field(default="https://api.example.com")

# Implementation
class MyCustomTool(Function):
    def __init__(self, config: MyToolConfig):
        super().__init__(config)
        self.api_key = config.api_key
        self.endpoint = config.endpoint

    @property
    def input_schema(self):
        return MyToolInput

    @property
    def name(self):
        return "my_custom_tool"

    @property
    def description(self):
        return "Custom tool for specific functionality"
```

```python
    def __call__(self, query: str, limit: int = 10) -> str:
        # Implementation
        result = self.call_api(query, limit)
        return f"Found {len(result)} results"

    def call_api(self, query, limit):
        # API call logic
        pass
```

YAML Usage

```yaml
functions:
  my_tool:
    _type: my_custom_tool
    api_key: ${MY_API_KEY}
    endpoint: "https://api.example.com"

workflow:
  _type: react_agent
  tool_names: [my_tool]
  llm_name: nim_llm
```

Best Practices

1. Agent Selection Decision Tree

```
Start
  |
```

```
├── Need complex reasoning between steps?
│   ├── Yes → Need token efficiency?
│   │   ├── Yes → ReWOO Agent
│   │   └── No → ReAct Agent
│   │
│   └── No → Have reasoning LLM?
│      ├── Yes → Reasoning Agent (with Tool Calling base)
│      └── No → Tool Calling Agent
│
└── Need debugging visibility? → ReAct Agent
```

2. Tool Design Principles

1. **Single Responsibility**: One tool = one clear purpose
2. **Descriptive Names**: `calculate_mortgage_payment` > `calc`
3. **Rich Descriptions**: Include examples in descriptions
4. **Schema Validation**: Always define input schemas
5. **Error Messages**: Return actionable error messages

3. Prompt Engineering

Good Tool Description:

```
python
```

```
description = """
Calculate monthly mortgage payment including principal and interest.

Parameters:
- principal: Loan amount in dollars (e.g., 300000)
- interest_rate: Annual interest rate as percentage (e.g., 3.5)
- years: Loan term in years (e.g., 30)

Returns: Monthly payment amount in dollars

Example: calculate_mortgage(300000, 3.5, 30) → $1347.13
"""
```

Bad Tool Description:

```python
description = "Calculates mortgage"  # Too vague!
```

4. Configuration Management

```yaml
```

```yaml
# Use environment variables
llms:
  production_llm:
    _type: nim_llm
    model_name: ${LLM_MODEL_NAME}
    api_key: ${LLM_API_KEY}
    endpoint: ${LLM_ENDPOINT}

# Use includes for modularity
functions:
  _include:
    - tools/math_tools.yml
    - tools/search_tools.yml
    - tools/api_tools.yml

# Environment-specific configs
workflow:
  _type: react_agent
  tool_names: ${TOOL_NAMES}  # Different per environment
  verbose: ${DEBUG_MODE}
```

5. Performance Optimization

```
yaml
```

```yaml
# Token optimization
workflow:
  _type: rewoo_agent  # Most token-efficient
  max_history: 5      # Limit context
  include_tool_input_schema_in_tool_description: false  # Reduce prompt size

# Speed optimization
workflow:
  _type: tool_calling_agent  # Fastest execution
  max_iterations: 5          # Prevent long chains
  handle_tool_errors: false   # Fail fast

# Quality optimization
workflow:
  _type: reasoning_agent
  llm_name: best_llm
  augmented_fn: react_agent
  verbose: true
```

Common Patterns & Examples

Pattern 1: RAG Agent

yaml

```yaml
functions:
  vector_search:
    _type: vector_search
    embedding_model: text-embedding-ada-002
    index_name: knowledge_base

  reranker:
    _type: cross_encoder_reranker
    model_name: cross-encoder/ms-marco-MiniLM-L-6-v2

  document_qa:
    _type: react_agent
    tool_names: [vector_search, reranker]
    llm_name: gpt-4
    system_prompt: |
      You are a QA agent. Use vector_search to find relevant documents,
      then reranker to select the best ones. Answer based only on retrieved content.

workflow:
  _type: reasoning_agent
  llm_name: deepseek_r1
  augmented_fn: document_qa
```

Pattern 2: Code Agent

```
yaml
```

```yaml
functions:
  code_interpreter:
    _type: python_repl
    sandbox: true

  linter:
    _type: code_linter
    languages: [python, javascript]

  test_runner:
    _type: pytest_runner

  code_agent:
    _type: react_agent
    tool_names: [code_interpreter, linter, test_runner]
    llm_name: code_llm
    system_prompt: |
      Generate code, lint it, run tests. Iterate until all tests pass.

workflow:
  _type: reasoning_agent
  llm_name: reasoning_llm
  augmented_fn: code_agent
```

Pattern 3: Multi-Modal Agent

yaml

```yaml
functions:
  image_analyzer:
    _type: vision_api
    model: gpt-4-vision

  image_generator:
    _type: dalle_api
    model: dall-e-3

  ocr_tool:
    _type: ocr_engine

  multimodal_agent:
    _type: tool_calling_agent
    tool_names: [image_analyzer, image_generator, ocr_tool]
    llm_name: vision_llm

workflow:
  _type: reasoning_agent
  llm_name: reasoning_llm
  augmented_fn: multimodal_agent
```

Conclusion

NVIDIA Agent Intelligence Toolkit provides:

1. **Four Agent Types** for different use cases
2. **Full MCP Support** for distributed architectures
3. **Framework Agnostic** design

4. **Production Ready** with observability and profiling
5. **Highly Composable** function-based architecture

Quick Selection Guide:

- **ReAct**: Complex, unpredictable workflows
- **Reasoning**: High-quality planning with reasoning LLMs
- **ReWOO**: Token-efficient, predictable workflows
- **Tool Calling**: Fast, structured tasks

Next Steps:

1. Install: `uv pip install aiqtoolkit`
2. Choose agent type based on your use case
3. Define tools in YAML
4. Run: `aiq run --config_file config.yml --input "query"`
5. Iterate and optimize

For full documentation: https://docs.nvidia.com/aiqtoolkit/latest/