

# Tutorial 02: Function Tools - Give Your Agent Superpowers

---

**Difficulty:** beginner

**Reading Time:** 45 minutes

**Tags:** beginner, tools, functions, python

**Description:** Learn how to add custom Python functions as tools to make your agent capable of calculations, data processing, and problem-solving.

# Tutorial 02: Function Tools - Give Your Agent Superpowers

---



**Working Implementation:** See the complete, tested code at

`tutorial_implementation/tutorial02/` ([https://github.com/raphaelmansuy/adk\\_training/tree/main/tutorial\\_implementation/tutorial02/](https://github.com/raphaelmansuy/adk_training/tree/main/tutorial_implementation/tutorial02/))

## Overview

---

Transform your agent from a conversationalist into a problem-solver! In this tutorial, you'll learn how to give your agent custom abilities by adding Python functions as tools. Your agent will automatically decide when to use these tools based on user requests.

## Prerequisites

---

- **Completed Tutorial 01** - You should have a working hello agent

- **Python functions knowledge** - Understanding of function definitions, parameters, and return values
- **Installed ADK** - `pip install google-adk`
- **API key configured** - From Tutorial 01

## Core Concepts

---

### | Function Tools

**Function tools** are regular Python functions that you give to your agent. The agent can call these functions when it needs to perform specific tasks. ADK automatically:

- Reads your function signature (parameters, types, defaults)
- Reads your docstring (what the function does)
- Generates a schema the LLM can understand
- Lets the LLM decide WHEN to call your function

### | Tool Discovery

The **LLM is smart** - it reads your function's name, docstring, and parameters, then decides if it should call that function based on the user's request. You don't manually trigger tools!

### | Return Values

Tools should return **dictionaries** with:

- `"status"`: `"success"` or `"error"`
- `"report"`: The actual result or error message

This helps the LLM understand what happened.

## Use Case

---

We're building a **Personal Finance Assistant** that can:

- Calculate compound interest for savings
- Compute monthly loan payments
- Determine how much to save monthly for a goal
- Explain financial concepts

This demonstrates real-world tool use - calculations the LLM can't do accurately on its own!

## Step 1: Create Project Structure

---

Create a new directory for the finance assistant:

```
mkdir finance_assistant
cd finance_assistant
touch __init__.py agent.py .env
```

Copy your `.env` file from Tutorial 01, or create it with your API key.

## Step 2: Set Up Package Import

---

**finance\_assistant/init.py**

```
from . import agent
```

## Step 3: Define Tool Functions

---

Now the fun part - create Python functions that do the actual calculations!

**finance\_assistant/agent.py**

```

from __future__ import annotations

from google.adk.agents import Agent

# Tool 1: Calculate compound interest
def calculate_compound_interest(
    principal: float,
    annual_rate: float,
    years: int,
    compounds_per_year: int = 1
) -> dict:
    """
    Calculate compound interest for savings or investments.

    This function computes how much an initial investment will grow to
    over time with compound interest. It uses the standard compound interest
    formula:  $A = P(1 + r/n)^{nt}$ 

    Args:
        principal: Initial investment amount (e.g., 10000 for $10,000)
        annual_rate: Annual interest rate as decimal (e.g., 0.06 for 6%)
        years: Number of years to compound
        compounds_per_year: How often interest compounds per year (default: 1)

    Returns:
        Dict with calculation results and formatted report

    Example:
        >>> calculate_compound_interest(10000, 0.06, 5)
        {
            'status': 'success',
            'final_amount': 13488.50,
            'interest_earned': 3488.50,
            'report': 'After 5 years at 6% annual interest...'
        }
    """
    try:
        # Validate inputs
        if principal <= 0:
            return {
                'status': 'error',
                'error': 'Principal must be positive',
                'report': 'Error: Investment principal must be greater than zero'
            }

        if annual_rate < 0 or annual_rate > 1:

```

```

        return {
            'status': 'error',
            'error': 'Invalid interest rate',
            'report': 'Error: Annual interest rate must be between 0 and 1
        }

    if years <= 0:
        return {
            'status': 'error',
            'error': 'Invalid time period',
            'report': 'Error: Investment period must be positive.'
        }

    # Calculate compound interest
    rate_per_period = annual_rate / compounds_per_year
    total_periods = years * compounds_per_year

    final_amount = principal * (1 + rate_per_period) ** total_periods
    interest_earned = final_amount - principal

    # Format human-readable report
    report = (
        f"After {years} years at {annual_rate*100:.1f}% annual interest "
        f"(compounded {compounds_per_year} times per year), "
        f"your ${principal:,.0f} investment will grow to "
        f"${final_amount:,.2f}. That's ${interest_earned:,.2f} in interest
    )

    return {
        'status': 'success',
        'final_amount': round(final_amount, 2),
        'interest_earned': round(interest_earned, 2),
        'report': report
    }

except Exception as e:
    return {
        'status': 'error',
        'error': str(e),
        'report': f'Error calculating compound interest: {str(e)}'
    }

# Tool 2: Calculate loan payments
def calculate_loan_payment(
    loan_amount: float,
    annual_rate: float,
    years: int

```

```

) -> dict:
    """Calculate monthly loan payments using the standard amortization formula

    This function computes the monthly payment required to pay off a loan
    over a specified period at a given interest rate. It uses the formula:
     $M = P[r(1+r)^n]/[(1+r)^n-1]$  where  $r$  is monthly rate and  $n$  is months.

    Args:
        loan_amount: Total loan amount (e.g., 300000 for $300,000)
        annual_rate: Annual interest rate as decimal (e.g., 0.045 for 4.5%)
        years: Loan term in years

    Returns:
        Dict with payment calculation results and formatted report

    Example:
        >>> calculate_loan_payment(300000, 0.045, 30)
        {
            'status': 'success',
            'monthly_payment': 1520.06,
            'total_paid': 547221.60,
            'total_interest': 247221.60,
            'report': 'For a $300,000 loan at 4.5% over 30 years...'
        }
    """
    try:
        # Validate inputs
        if loan_amount <= 0:
            return {
                'status': 'error',
                'error': 'Invalid loan amount',
                'report': 'Error: Loan amount must be positive.'
            }

        if annual_rate < 0 or annual_rate > 1:
            return {
                'status': 'error',
                'error': 'Invalid interest rate',
                'report': 'Error: Annual interest rate must be between 0 and 1
                    (e.g., 0.045 for 4.5%).'
            }

        if years <= 0:
            return {
                'status': 'error',
                'error': 'Invalid loan term',
                'report': 'Error: Loan term must be positive.'
            }

```

```

    }

    # Convert to monthly calculations
    monthly_rate = annual_rate / 12
    total_months = years * 12

    # Handle zero interest rate case
    if monthly_rate == 0:
        monthly_payment = loan_amount / total_months
        total_paid = loan_amount
        total_interest = 0
    else:
        # Standard loan payment formula
        monthly_payment = loan_amount * (
            monthly_rate * (1 + monthly_rate) ** total_months
        ) / ((1 + monthly_rate) ** total_months - 1)

        total_paid = monthly_payment * total_months
        total_interest = total_paid - loan_amount

    # Format human-readable report
    report = (
        f"For a ${loan_amount:,.0f} loan at {annual_rate*100:.1f}% interes
        f"over {years} years, your monthly payment will be "
        f"${monthly_payment:,.2f}. Over the life of the loan, you'll pay "
        f"${total_paid:,.2f} total, with ${total_interest:,.2f} being inte
    )

    return {
        'status': 'success',
        'monthly_payment': round(monthly_payment, 2),
        'total_paid': round(total_paid, 2),
        'total_interest': round(total_interest, 2),
        'report': report
    }

except Exception as e:
    return {
        'status': 'error',
        'error': str(e),
        'report': f'Error calculating loan payment: {str(e)}'
    }

# Tool 3: Calculate savings needed
def calculate_monthly_savings(
    target_amount: float,
    years: int,

```

```

    annual_return: float = 0.05
) -> dict:
    """Calculate monthly savings needed to reach a financial goal.

    This function determines how much you need to save each month to reach
    a savings goal, assuming compound growth at a specified annual return.
    It uses the present value of annuity formula rearranged for payment amount

    Args:
        target_amount: Target savings amount (e.g., 50000 for $50,000)
        years: Number of years to save
        annual_return: Expected annual return as decimal (default: 0.05 for 5%)

    Returns:
        Dict with savings calculation results and formatted report

    Example:
        >>> calculate_monthly_savings(50000, 3, 0.05)
        {
            'status': 'success',
            'monthly_savings': 1315.07,
            'total_contributed': 47342.52,
            'interest_earned': 2657.48,
            'report': 'To reach $50,000 in 3 years with 5% annual return...'
        }
    """
    try:
        # Validate inputs
        if target_amount <= 0:
            return {
                'status': 'error',
                'error': 'Invalid target amount',
                'report': 'Error: Savings target must be positive.'
            }

        if years <= 0:
            return {
                'status': 'error',
                'error': 'Invalid time period',
                'report': 'Error: Savings period must be positive.'
            }

        if annual_return < 0:
            return {
                'status': 'error',
                'error': 'Invalid return rate',
                'report': 'Error: Annual return rate cannot be negative.'
            }

```



```

    }

    # Convert to monthly calculations
    monthly_return = annual_return / 12
    total_months = years * 12

    # Handle zero return case
    if monthly_return == 0:
        monthly_savings = target_amount / total_months
        total_contributed = target_amount
        interest_earned = 0
    else:
        # Correct formula for monthly savings to reach future value
        # PMT = FV * (r / ((1 + r)^n - 1)) where r is monthly rate, n is m
        monthly_savings = target_amount * (
            monthly_return / ((1 + monthly_return) ** total_months - 1)
        )

        total_contributed = monthly_savings * total_months
        # Calculate actual future value to verify
        future_value = 0
        for month in range(1, total_months + 1):
            future_value += monthly_savings * (1 + monthly_return) ** (tot
            interest_earned = future_value - total_contributed

    # Format human-readable report
    report = (
        f"To reach ${target_amount:,.0f} in {years} years with a "
        f"{annual_return*100:.1f}% annual return, you need to save "
        f"${monthly_savings:,.2f} per month. You'll contribute "
        f"${total_contributed:,.2f} total, with the rest coming from inves
    )

    return {
        'status': 'success',
        'monthly_savings': round(monthly_savings, 2),
        'total_contributed': round(total_contributed, 2),
        'interest_earned': round(interest_earned, 2),
        'report': report
    }

except Exception as e:
    return {
        'status': 'error',
        'error': str(e),
        'report': f'Error calculating monthly savings: {str(e)}'
    }

```

```
# Define the agent with all tools
root_agent = Agent(
    name="finance_assistant",
    model="gemini-2.0-flash",
    description="""
    A financial calculation assistant that can help with:
    - Compound interest calculations for investments
    - Loan payment calculations for mortgages or other loans
    - Monthly savings calculations to reach financial goals

    I can perform multiple calculations simultaneously for comparison purposes
    All calculations include detailed explanations and formatted reports.
    """,
    instruction=(
        "You are a helpful personal finance assistant. You can help users with\n"
        "- Calculating compound interest for savings and investments\n"
        "- Computing monthly payments for loans (mortgages, car loans, etc.)\n"
        "- Determining how much to save monthly to reach financial goals\n"
        "\n"
        "When users ask financial questions:\n"
        "1. Use the appropriate calculation tool\n"
        "2. Explain the results in simple terms\n"
        "3. Provide context and advice when relevant\n"
        "4. Be encouraging and positive about their financial planning!\n"
        "\n"
        "You are NOT a licensed financial advisor - remind users to consult pr
    ),
    tools=[calculate_compound_interest, calculate_loan_payment, calculate_mont
)
```

## Code Breakdown

### Function Signature Best Practices:

1. **Type hints** - `principal: float`, `years: int` - tell the LLM what types to use
2. **Clear parameter names** - `annual_rate` not just `rate`
3. **Defaults for optional params** - `compounds_per_year: int = 12`
4. **Comprehensive docstring** - explain WHAT the function does and WHEN to use it

### Return Value Pattern:

```
return {  
  "status": "success", # or "error"  
  "report": "Human-readable result" # or "error_message" for errors  
}
```

This structured format helps the LLM understand what happened and generate better responses.

### Tool Registration:

Notice we just pass the functions directly to `tools=[...]` - ADK automatically converts them to tools!

## Step 4: Run Your Finance Assistant

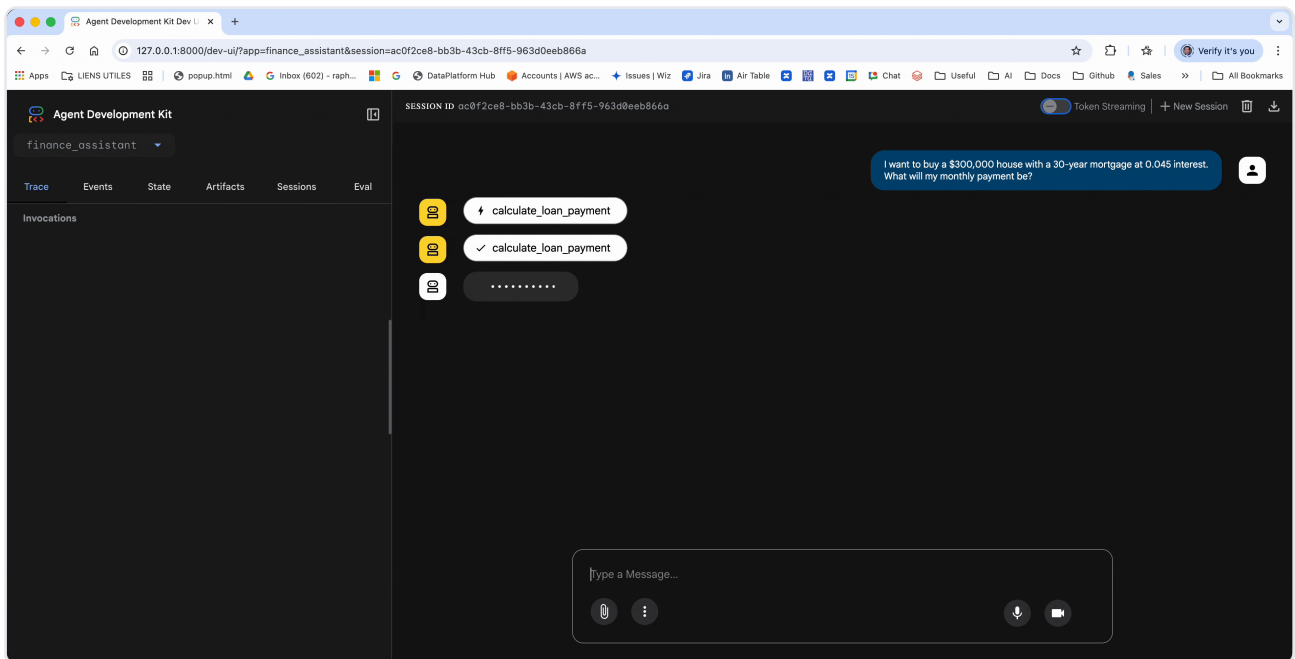
Navigate to the parent directory and launch the Dev UI:

```
cd .. # Go to parent of finance_assistant/  
adk web
```

Open `http://localhost:8000` and select "finance\_assistant" from the dropdown.

## | Demo in Action

Here's what your finance assistant looks like in action:



## Alternative: Parallel Execution Demo

For an advanced demo showcasing ADK's automatic parallel tool execution, try the parallel demo:

```
cd .. # Go to parent of finance_assistant/  
make parallel-demo
```

This runs the same financial tools but demonstrates how ADK automatically executes multiple tools simultaneously when Gemini requests them in a single turn. Perfect for comparing multiple investment options or analyzing different loan scenarios!

## Try These Prompts

### Savings Calculation:

If I invest \$10,000 at 0.06 annual interest for 5 years, how much will I have?

### Loan Payment:

I want to buy a \$300,000 house with a 30-year mortgage at 0.045 interest. What will my monthly payment be?

### Savings Goal:

I want to save \$50,000 for a down payment in 3 years.  
How much should I save each month?

### Mixed Question (LLM decides NOT to use tools):

What's the difference between a Roth IRA and a traditional IRA?

Notice on this last one - the agent WON'T call any tools because none of them help answer that question. The LLM will just use its knowledge!

## Understanding Tool Execution

Open the **Events tab** in the Dev UI after sending a message. You'll see:

1. **User Message:** Your original question
2. **Function Call Event:** Shows which tool was called with what parameters
3. **Function Response Event:** Shows what the tool returned
4. **Model Response:** The final answer combining tool results with explanation

This is incredibly powerful for debugging - you can see EXACTLY when and how tools are used!

## Expected Behavior

### Example 1: Savings Calculation

User: If I invest \$10,000 at 0.06 annual interest for 5 years, how much will I

Agent: Let me calculate that for you!

[Tool Call: calculate\_compound\_interest(principal=10000, annual\_rate=0.06, yea

Based on the calculation, if you invest \$10,000 at 6% annual interest compound monthly for 5 years, you'll have approximately \$13,488.50. That means you'll e \$3,488.50 in interest - not bad! This assumes the interest compounds monthly, is common for savings accounts and many investment vehicles.

## Example 2: No Tool Needed

User: Should I invest in stocks or bonds?

Agent: That's a great question about investment strategy! The choice between s and bonds depends on several factors: your age, risk tolerance, financial goal time horizon...

[No tools called - the LLM just uses its knowledge]

## How It Works (Behind the Scenes)

1. **User sends message** → ADK receives it
2. **ADK packages the message** with:
  3. Your agent's instructions
  4. Tool schemas (generated from your function signatures)
5. **Gemini (LLM) decides**: "Do I need a tool for this?"
6. **YES** → Returns a function call request
7. **NO** → Returns a text response
8. **If tool needed**, ADK:
  9. Executes your Python function with the parameters Gemini provided
  10. Gets the return value
  11. Sends it back to Gemini
12. **Gemini generates final response** using the tool result

**You never manually call tools** - the LLM does it automatically!



# Advanced: Parallel Tool Calling

**Source:** `google/adk/flows/llm_flows/functions.py`

One of ADK's most powerful features is **automatic parallel tool execution**. When the LLM requests multiple tools in a single turn, ADK executes them **simultaneously** using `asyncio.gather()` - dramatically improving performance!

## How It Works

When Gemini decides to call multiple tools, instead of executing them one-by-one:

```
# ❌ Sequential execution (slow)
result1 = await tool1() # Wait...
result2 = await tool2() # Wait...
result3 = await tool3() # Wait...
# Total time: ~6 seconds

# ✅ Parallel execution (fast) - ADK does this automatically!
results = await asyncio.gather(tool1(), tool2(), tool3())
# Total time: ~2 seconds (limited by slowest tool)
```

**You don't need to do anything** - ADK handles this automatically! Just define your tools normally.

## Real-World Example: Multi-City Financial Planning

Let's extend our finance assistant to handle parallel calculations:

```

from __future__ import annotations
import asyncio
from google.adk.agents import Agent

def calculate_compound_interest(
    principal: float,
    annual_rate: float,
    years: int,
    compounds_per_year: int = 12
) -> dict:
    """Calculate compound interest for savings or investments.

    Args:
        principal: The initial amount invested (in dollars)
        annual_rate: The annual interest rate as a percentage (e.g., 5.5 for 5.5%)
        years: Number of years to calculate for
        compounds_per_year: How many times per year interest compounds (default 12)

    Returns:
        dict: Dictionary with status and calculation results
    """
    # Add simulated delay to show parallel execution benefit
    import time
    time.sleep(0.5) # Simulate API call or heavy computation

    rate_decimal = annual_rate / 100
    final_amount = principal * (1 + rate_decimal / compounds_per_year) ** (years * compounds_per_year)
    interest_earned = final_amount - principal

    return {
        "status": "success",
        "report": (
            f"Investment: ${principal:,.2f}\n"
            f"Final amount: ${final_amount:,.2f}\n"
            f"Interest earned: ${interest_earned:,.2f}"
        )
    }

def calculate_loan_payment(
    loan_amount: float,
    annual_rate: float,
    years: int
) -> dict:
    """Calculate monthly payment for a loan."""
    import time
    time.sleep(0.5) # Simulate processing

```



```

monthly_rate = (annual_rate / 100) / 12
num_payments = years * 12

if monthly_rate == 0:
    monthly_payment = loan_amount / num_payments
else:
    monthly_payment = loan_amount * (monthly_rate * (1 + monthly_rate)**nu

return {
    "status": "success",
    "report": f"Monthly payment: ${monthly_payment:,.2f}"
}

def calculate_monthly_savings(
    target_amount: float,
    years: int,
    annual_return: float = 5.0
) -> dict:
    """Calculate monthly savings needed to reach a goal."""
    import time
    time.sleep(0.5) # Simulate calculation

    months = years * 12
    monthly_rate = (annual_return / 100) / 12

    if monthly_rate == 0:
        monthly_savings = target_amount / months
    else:
        monthly_savings = target_amount / (((1 + monthly_rate)**months - 1) /

    return {
        "status": "success",
        "report": f"Save ${monthly_savings:,.2f} per month"
    }

parallel_finance_agent = Agent(
    name="parallel_finance_assistant",
    model="gemini-2.5-flash", # Supports parallel tool calling!
    description="Financial assistant with parallel computation",
    instruction=(
        "You are a financial planning assistant. When users ask about multiple
        "scenarios or calculations, call ALL necessary tools at once to be eff
        "For example, if comparing investment options, call the calculation to
        "EACH option simultaneously."
    ),
    tools=[

```

```

        calculate_compound_interest,
        calculate_loan_payment,
        calculate_monthly_savings
    ]
)

```

## Try This Prompt (Triggers Parallel Execution)

Compare these three investment options for me:

1. \$10,000 at 0.05 for 10 years
2. \$15,000 at 0.04 for 10 years
3. \$12,000 at 0.06 for 10 years

### What happens:

1. Gemini recognizes it needs to call `calculate_compound_interest` THREE times
2. ADK receives THREE `FunctionCall` objects from Gemini
3. ADK executes all three **simultaneously** with `asyncio.gather()`
4. All results come back in ~0.5s instead of ~1.5s (sequential)
5. Gemini receives all results and generates comparative analysis

## Performance Comparison

### Sequential Execution (if you did it manually):

```

# ❌ Slow approach (not how ADK works)
result1 = calculate_compound_interest(10000, 0.05, 10) # 0.5s
result2 = calculate_compound_interest(15000, 0.04, 10) # 0.5s
result3 = calculate_compound_interest(12000, 0.06, 10) # 0.5s
# Total: ~1.5 seconds

```

### Parallel Execution (ADK automatic):

```
# ✓ Fast - ADK does this for you!
results = await asyncio.gather(
    calculate_compound_interest(10000, 0.05, 10),
    calculate_compound_interest(15000, 0.04, 10),
    calculate_compound_interest(12000, 0.06, 10)
)
# Total: ~0.5 seconds (limited by slowest tool)
```

**Speedup:** 3x faster for 3 parallel tools!

## Example Output

User: Compare these three investment options for me:

1. \$10,000 at 0.05 for 10 years
2. \$15,000 at 0.04 for 10 years
3. \$12,000 at 0.06 for 10 years

Agent: Let me calculate all three options for you...

[Tool Calls - EXECUTED SIMULTANEOUSLY]:

- calculate\_compound\_interest(principal=10000, annual\_rate=0.05, years=10)
- calculate\_compound\_interest(principal=15000, annual\_rate=0.04, years=10)
- calculate\_compound\_interest(principal=12000, annual\_rate=0.06, years=10)

Great question! Here's how your three investment options compare:

**\*\*Option 1\*\*:** \$10,000 at 5% for 10 years

- Final amount: \$16,470.09
- Interest earned: \$6,470.09

**\*\*Option 2\*\*:** \$15,000 at 4% for 10 years

- Final amount: \$22,280.97
- Interest earned: \$7,280.97

**\*\*Option 3\*\*:** \$12,000 at 6% for 10 years

- Final amount: \$21,791.23
- Interest earned: \$9,791.23

Option 3 gives you the highest return (\$9,791.23 in interest), even with a low principal than Option 2. That extra 2% rate makes a big difference over 10 years.

## | When Does Parallel Execution Happen?

Parallel execution occurs when:

- ✓ **Multiple tool calls in single turn** - Gemini decides to call 2+ tools at once
- ✓ **Tools are independent** - Results don't depend on each other
- ✓ **Model supports parallel calling** - Gemini 2.5-flash, 2.5-pro, 2.0-flash support this

**Does NOT happen when:**

- ✗ **Sequential dependencies** - Tool B needs result from Tool A
- ✗ **Single tool call** - Only one tool invoked
- ✗ **Manual sequential instructions** - You explicitly tell the model to do things step-by-step

## | Optimizing for Parallel Execution

✓ **DO: Design independent tools**

```
# Good - These can run in parallel
def get_weather(city: str): ...
def get_exchange_rate(currency: str): ...
def get_stock_price(symbol: str): ...

# User: "What's the weather in Tokyo, EUR/USD rate, and AAPL stock price?"
# → All 3 execute simultaneously!
```

✗ **DON'T: Create dependencies**

```
# Bad - These create a dependency chain
def search_database(query: str) -> dict:
    """Find database records."""
    return {"status": "success", "record_id": "123"}

def fetch_record_details(record_id: str) -> dict:
    """Get full details for a record (needs record_id first)."""
    return {"status": "success", "details": "..."}

# These MUST run sequentially - can't parallelize
```

## Verification: Check the Events Tab

Open the Dev UI Events tab after sending a multi-tool query. Look for:

```
[FunctionCall] calculate_compound_interest(principal=10000, ...)
[FunctionCall] calculate_compound_interest(principal=15000, ...)
[FunctionCall] calculate_compound_interest(principal=12000, ...)

[FunctionResponse] result for 10000
[FunctionResponse] result for 15000
[FunctionResponse] result for 12000
```

Notice all `FunctionCall` events are emitted **before** any `FunctionResponse` - proof they executed in parallel!

## Source Code Reference

The parallel execution implementation is in `google/adk/flows/llm_flows/functions.py`:

```
# Simplified version of what ADK does internally
async def execute_function_calls(calls: list[FunctionCall]):
    """Execute multiple function calls in parallel."""

    tasks = [execute_single_function(call) for call in calls]
    results = await asyncio.gather(*tasks)

    return results
```

**You get this for free** - just define your tools normally!

## Performance Tips

1. **For I/O-bound tools** (API calls, database queries):
2. Parallel execution provides **massive speedup** (3-10x)
3. Each tool waits on network, not CPU
4. **For CPU-bound tools** (calculations, data processing):
5. Parallel execution still helps if tools are independent

6. Python GIL limits pure CPU parallelism, but asyncio scheduling still improves responsiveness
7. **Mixed workloads** (some I/O, some CPU):
8. I/O tools finish during CPU tool execution
9. Best of both worlds!

## Advanced Example: Multi-Source Data Aggregation

```
def get_market_data(symbol: str) -> dict:
    """Fetch stock market data (simulated API call)."""
    import time
    time.sleep(1.0) # Simulate API latency
    return {
        "status": "success",
        "report": f"{symbol}: $150.32 (+2.1%)"
    }

def get_company_news(symbol: str) -> dict:
    """Fetch latest news for a company (simulated API call)."""
    import time
    time.sleep(1.2) # Simulate API latency
    return {
        "status": "success",
        "report": f"{symbol} announces Q4 earnings beat"
    }

def get_analyst_ratings(symbol: str) -> dict:
    """Fetch analyst ratings (simulated API call)."""
    import time
    time.sleep(0.8) # Simulate API latency
    return {
        "status": "success",
        "report": f"{symbol}: 12 Buy, 3 Hold, 1 Sell"
    }

aggregator_agent = Agent(
    name="market_aggregator",
    model="gemini-2.5-flash",
    description="Aggregates market data from multiple sources",
    instruction="When asked about a stock, fetch ALL relevant data simultaneously",
    tools=[get_market_data, get_company_news, get_analyst_ratings]
)

# Query: "Tell me everything about AAPL"
# → All 3 tools execute in parallel (~1.2s total vs ~3s sequential)
```

## Sample Reference

Check out `contributing/samples/parallel_functions/agent.py` for a complete working example of parallel tool execution.

## Key Takeaways

- ✓ **Tools are just Python functions** - No special classes needed, just regular functions!
- ✓ **LLM decides when to use tools** - You don't manually trigger them. The LLM reads the docstring and decides.
- ✓ **Parallel execution is automatic** - When multiple tools are called, ADK runs them simultaneously via `asyncio.gather()`
- ✓ **Type hints are critical** - They tell the LLM what data types to use for parameters
- ✓ **Docstrings = tool descriptions** - Write clear docstrings explaining WHEN and HOW to use the tool
- ✓ **Return dicts with status** - Use `{"status": "success", "report": "..."}`  pattern for clarity
- ✓ **Default parameters = optional** - Functions with defaults can be called without those params
- ✓ **Events tab is your debugging friend** - See every tool call, parameter, and response (and verify parallel execution!)
- ✓ **Tools extend LLM capabilities** - Use tools for calculations, API calls, database queries - anything the LLM can't do alone
- ✓ **Design for independence** - Tools that don't depend on each other enable parallel execution and better performance

## Best Practices

### DO:

- Write descriptive function names ( `calculate_compound_interest` not `calc_int` )



- Include comprehensive docstrings
- Use type hints for all parameters
- Return structured dictionaries
- Handle errors gracefully
- Keep tools focused (one function = one task)

**DON'T:**

- Use generic names ( `process_data` , `do_stuff` )
- Rely on `*args` or `**kwargs` for LLM-facing parameters (they're ignored!)
- Return complex objects (stick to dicts, strings, numbers)
- Make tools that do too many things
- Forget to handle error cases

## Common Issues

---

**Problem:** "Tool not being called"

- **Check:** Is your docstring clear about WHEN to use the tool?
- **Check:** Does the function name match what the user is asking for?
- **Tip:** Look at Events tab - did Gemini even consider the tool?

**Problem:** "Wrong parameters passed"

- **Check:** Are your type hints correct?
- **Check:** Is your docstring describing parameters clearly?
- **Try:** Add examples in the docstring

**Problem:** "Tool returns error"

- **Check:** Add try/except blocks to catch errors
- **Return:** Error status dict instead of raising exceptions

## What We Built

---


You now have a finance assistant agent that:

- Performs accurate compound interest calculations
- Computes loan payments
- Plans savings goals
- Explains results in human-friendly language

And you learned HOW ADK tools work under the hood!

## Next Steps

---

 **Tutorial 03: OpenAPI Tools** - Connect to real web APIs (weather, stock prices, news, etc.)

 **Further Reading:**

- [Function Tools Documentation](https://google.github.io/adk-docs/tools/function-tools/) (https://google.github.io/adk-docs/tools/function-tools/)
- [Tool Performance \(Parallel Execution\)](https://google.github.io/adk-docs/tools/performance/) (https://google.github.io/adk-docs/tools/performance/)
- [Built-in Tools](https://google.github.io/adk-docs/tools/built-in-tools/) (https://google.github.io/adk-docs/tools/built-in-tools/)

## Exercises (Try On Your Own!)

---

1. **Add a budgeting tool** - Calculate if someone can afford something based on income
2. **Add debt payoff tool** - Calculate how long to pay off credit card debt
3. **Add retirement savings tool** - Estimate retirement savings needs
4. **Handle more edge cases** - What if someone enters negative numbers?

## Complete Code Reference

---

**finance\_assistant/init.py**

```
from . import agent
```

### finance\_assistant/.env

```
GOOGLE_GENAI_USE_VERTEXAI=FALSE  
GOOGLE_API_KEY=your-api-key-here
```

### finance\_assistant/agent.py

```
# See the complete implementation at tutorial_implementation/tutorial02/finance_assistant/agent.py  
# This file contains the full agent code with comprehensive error handling,  
# input validation, and detailed docstrings.
```

Congratulations! Your agent now has superpowers! 🚀💰

Generated on 2025-10-19 17:56:04 from 02\_function\_tools.md

Source: Google ADK Training Hub