Tutorial 13: Code Execution - Dynamic Python Code Generation

Difficulty: advanced

Reading Time: 1.5 hours

Tags: advanced, code-execution, python, calculations, data-analysis

Description: Enable your agents to write and execute Python code for calculations, data analysis, and complex computations using Gemini 2.0+'s built-in code execution capability.

Overview

Goal: Enable your agents to write and execute Python code for calculations, data analysis, and complex computations using Gemini 2.0+'s built-in code execution capability.

Prerequisites:

- Tutorial 01 (Hello World Agent)
- Tutorial 02 (Function Tools)
- Gemini 2.0+ model access

What You'll Learn:

- Using `BuiltInCodeExecutor` for code generation and execution
- Understanding model-side code execution (no local execution)
- Building data analysis agents
- Creating calculation assistants
- Handling code execution errors
- Best practices for code-based agents

Time to Complete: 40-55 minutes

🚀 Quick Start

The fastest way to get started is with our working implementation:

```
cd tutorial_implementation/tutorial13
make setup
make dev
```

Then open `http://localhost:8000` in your browser and select "code_calculator"!

Or explore the complete implementation: Tutorial 13 Implementation (https://github.com/raphaelmansuy/adk_training/tree/main/tutorial_implementation/tutorial13)

Why Code Execution Matters

AI models are excellent at reasoning but historically struggled with precise calculations. **Code execution** solves this by allowing models to:

- 🧮 **Perform Exact Calculations**: No approximation errors
- 📊 **Analyze Data**: Process arrays, statistics, transformations
- 🔬 **Solve Complex Problems**: Multi-step mathematical operations
- 📈 **Generate Visualizations**: Create charts and graphs (as code)
- ⚡ **Execute Algorithms**: Sort, search, optimize

Without Code Execution:

```
User: "What's the factorial of 50?"
Agent: "The factorial of 50 is approximately 3.04 × 10^64"
      ↑ Approximation, may be inaccurate
```

With Code Execution:

```
User: "What's the factorial of 50?"
Agent: [Generates and executes: math.factorial(50)]
       "The factorial of 50 is exactly: 30414093201713378043612608166064768844
       ↑ Exact answer via code execution
```

Building on Previous Tutorials

Code execution represents a **quantum leap** from the function tools you learned in Tutorial 02. Let's see how it builds on previous concepts:

From Tutorial 01: Hello World Agent

Tutorial 01 taught you basic agent structure:

```
# Tutorial 01 - Basic Agent
agent = Agent(
    model='gemini-2.0-flash',
    name='hello_agent',
    instruction='You are a helpful assistant.'
)
```

Tutorial 13 adds code execution capabilities:

```
# Tutorial 13 - Agent with Code Execution
agent = Agent(
    model='gemini-2.0-flash',
    name='calculator',
    instruction='You can write and execute Python code.',
    code_executor=BuiltInCodeExecutor()  # ← New capability
)
```

From Tutorial 02: Function Tools

Tutorial 02 showed how to create custom tools:

```python
# Tutorial 02 - Custom Function Tool
def calculate_square(x: float) -> float:
    """Calculate the square of a number."""
    return x * x

agent = Agent(
    model='gemini-2.0-flash',
    tools=[FunctionTool(calculate_square)]
)
```

Tutorial 13 enables **dynamic tool creation**:

```python
# Tutorial 13 - Dynamic Code Generation
agent = Agent(
    model='gemini-2.0-flash',
    code_executor=BuiltInCodeExecutor()
)

# Agent can now create ANY mathematical function on demand
result = runner.run("Create a function to calculate compound interest", agent=
# Agent generates and executes the exact code needed
```

Evolution Comparison

Aspect	Tutorial 02 (Function Tools)	Tutorial 13 (Code Execution)
Tool Creation	Pre-defined functions	Dynamic code generation
Flexibility	Limited to coded tools	Unlimited Python capabilities
Accuracy	Depends on implementation	Exact mathematical precision
Maintenance	Update code for new tools	Agent learns new capabilities
Use Cases	Specific business logic	Any computational task

Practical Example: Calculator Evolution

Before (Tutorial 02 style):

```python
# Limited to pre-built functions
def add_numbers(a: float, b: float) -> float:
    return a + b

def multiply_numbers(a: float, b: float) -> float:
    return a * b

agent = Agent(
    model='gemini-2.0-flash',
    tools=[FunctionTool(add_numbers), FunctionTool(multiply_numbers)]
)

# Can only do: 2+2=4, 3*5=15
```

After (Tutorial 13 style):

```python
# Unlimited computational power
agent = Agent(
    model='gemini-2.0-flash',
    code_executor=BuiltInCodeExecutor()
)

# Can do ANYTHING:
# - Matrix operations
# - Statistical analysis
# - Algorithm implementation
# - Complex financial calculations
# - Scientific computations
```

Real-World Impact

Tutorial 02 Agent: "I can add numbers and multiply them."

Tutorial 13 Agent: "I can solve differential equations, perform statistical analysis, implement machine learning algorithms, calculate orbital mechanics, analyze financial portfolios, and much more - all with mathematical precision."

1. BuiltInCodeExecutor Basics

What is BuiltInCodeExecutor?

`BuiltInCodeExecutor` enables Gemini 2.0+ models to **generate Python code and execute it internally** within the model environment. No local code execution happens - everything runs inside Google's infrastructure.

Source: `google/adk/code_executors/built_in_code_executor.py`

Basic Usage

```python
from google.adk.agents import Agent, Runner
from google.adk.code_executors import BuiltInCodeExecutor

# Create agent with code execution
agent = Agent(
    model='gemini-2.0-flash',  # Requires Gemini 2.0+
    name='code_executor',
    instruction='You can write and execute Python code to solve problems.',
    code_executor=BuiltInCodeExecutor()
)

runner = Runner()
result = runner.run(
    "Calculate the sum of all prime numbers between 1 and 100",
    agent=agent
)

print(result.content.parts[0].text)
```

Output:

```
Let me calculate that using Python:

[Code executed:]
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

primes = [n for n in range(1, 101) if is_prime(n)]
sum(primes)

[Result:] 1060

The sum of all prime numbers between 1 and 100 is **1060**.
```

How It Works

Step-by-Step Process:

1. **User Query** → Model receives calculation request
2. **Code Generation** → Model writes Python code
3. **Code Execution** → Code runs in model environment (Google's infrastructure)
4. **Result Integration** → Execution result incorporated into response
5. **Final Answer** → Complete answer with explanation

Internal Implementation:

```python
# Simplified from built_in_code_executor.py
class BuiltInCodeExecutor(BaseCodeExecutor):
    def process_llm_request(self, llm_request: LlmRequest):
        """Add code execution tool to request."""
        llm_request.tools.append(
            types.Tool(code_execution=types.ToolCodeExecution())
        )
        return llm_request
```

Model Compatibility

```python
# ✔️ Works with Gemini 2.0+
agent = Agent(
    model='gemini-2.0-flash',
    code_executor=BuiltInCodeExecutor()
)

agent = Agent(
    model='gemini-2.0-flash-exp',
    code_executor=BuiltInCodeExecutor()
)

# ❌ Raises error with Gemini 1.x
agent = Agent(
    model='gemini-1.5-flash',
    code_executor=BuiltInCodeExecutor()
)
# Error: Code execution requires Gemini 2.0+
```

2. Code Execution Capabilities

Mathematical Calculations

```python
from google.adk.agents import Agent, Runner
from google.adk.code_executors import BuiltInCodeExecutor

math_agent = Agent(
    model='gemini-2.0-flash',
    name='mathematician',
    instruction='Solve mathematical problems using Python code.',
    code_executor=BuiltInCodeExecutor()
)

runner = Runner()

# Complex calculation
result = runner.run(
    "Calculate e^(π*i) + 1 and explain why the result is significant",
    agent=math_agent
)
print(result.content.parts[0].text)
```

Expected Output:

```
Let me calculate this using Python's complex number support:

[Code:]
import cmath
import math

result = cmath.exp(math.pi * 1j) + 1
print(f"Result: {result}")
print(f"Real part: {result.real}")
print(f"Imaginary part: {result.imag}")

[Output:]
Result: 1.2246467991473532e-16j
Real part: 1.2246467991473532e-16
Imaginary part: 0.0

The result is essentially **0** (the tiny number is due to floating-point prec

This is **Euler's Identity**: e^(πi) + 1 = 0

It's considered one of the most beautiful equations in mathematics because it
connects five fundamental constants: e, π, i, 1, and 0.
```

Data Analysis

```
data_analyst = Agent(
    model='gemini-2.0-flash',
    name='data_analyst',
    instruction='Analyze data using Python. Calculate statistics and identify
    code_executor=BuiltInCodeExecutor()
)

result = runner.run(
    """
Analyze this sales data and provide insights:
Month | Sales
Jan   | 15000
Feb   | 18000
Mar   | 16500
Apr   | 22000
May   | 21000
Jun   | 25000
    """,
    agent=data_analyst
)
```

Expected Output:

```
Let me analyze this sales data:

[Code:]
import statistics

sales = [15000, 18000, 16500, 22000, 21000, 25000]
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']

# Calculate statistics
mean = statistics.mean(sales)
median = statistics.median(sales)
stdev = statistics.stdev(sales)
total = sum(sales)

# Calculate growth
growth = [(sales[i] - sales[i-1]) / sales[i-1] * 100
          for i in range(1, len(sales))]

# Find best month
best_month_idx = sales.index(max(sales))

print(f"Mean: ${mean:,.2f}")
print(f"Median: ${median:,.2f}")
print(f"Std Dev: ${stdev:,.2f}")
print(f"Total: ${total:,}")
print(f"Best month: {months[best_month_idx]}")
print(f"Average growth: {statistics.mean(growth):.1f}%")

[Output:]
Mean: $19,583.33
Median: $19,500.00
Std Dev: $3,804.40
Total: $117,500
Best month: Jun
Average growth: 9.2%

**Key Insights:**
- Strong upward trend with 9.2% average monthly growth
- June showed best performance at $25,000
- Consistent growth from April onwards
- Recommendation: Investigate what drove April-June surge
```

Algorithm Implementation

```
algo_agent = Agent(
    model='gemini-2.0-flash',
    name='algorithm_expert',
    instruction='Implement and demonstrate algorithms using Python.',
    code_executor=BuiltInCodeExecutor()
)

result = runner.run(
    "Implement binary search and find the position of 42 in [1, 5, 12, 23, 42,
    agent=algo_agent
)
```

Expected Output:

```
[Code:]
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1

arr = [1, 5, 12, 23, 42, 67, 89, 99]
target = 42
position = binary_search(arr, target)

print(f"Found {target} at index {position}")

[Result:] Found 42 at index 4

Binary search found **42 at index 4** (the 5th element, zero-indexed).

The algorithm made only 3 comparisons instead of checking all 8 elements
linearly, demonstrating O(log n) efficiency.
```

3. Real-World Example: Financial Calculator

Let's build a comprehensive financial calculator agent.

Complete Implementation

```
"""
Financial Calculator Agent
Uses code execution for accurate financial calculations.
"""

import asyncio
import os
from google.adk.agents import Agent, Runner
from google.adk.code_executors import BuiltInCodeExecutor
from google.genai import types

# Environment setup
os.environ['GOOGLE_GENAI_USE_VERTEXAI'] = '1'
os.environ['GOOGLE_CLOUD_PROJECT'] = 'your-project-id'
os.environ['GOOGLE_CLOUD_LOCATION'] = 'us-central1'

# Create financial calculator agent
financial_calculator = Agent(
    model='gemini-2.0-flash',
    name='financial_calculator',
    description='Expert financial calculator with Python code execution',
    instruction="""
You are a financial calculator expert. For all calculations:

1. Write Python code to calculate exact values
2. Show the code you're running
3. Explain the formulas used
4. Present results clearly with $ formatting
5. Provide financial interpretation

Available calculations:
- Compound interest
- Present/Future value
- Loan amortization
- Investment returns (ROI, CAGR)
- Retirement planning
- Net present value (NPV)
- Internal rate of return (IRR)

Always execute code for accuracy. Never approximate.
    """.strip(),
    code_executor=BuiltInCodeExecutor(),
    generate_content_config=types.GenerateContentConfig(
        temperature=0.1,  # Very low for financial accuracy
        max_output_tokens=2048
    )
)

async def calculate_financial(query: str):
    """Run financial calculation."""

    print(f"\n{'='*70}")
    print(f"QUERY: {query}")
    print(f"{'='*70}\n")

    runner = Runner()
    result = await runner.run_async(query, agent=financial_calculator)

    print("💰 CALCULATION:\n")
    print(result.content.parts[0].text)
    print(f"\n{'='*70}\n")

async def main():
    """Run financial calculation examples."""

    # Example 1: Compound Interest
    await calculate_financial("""
If I invest $10,000 at 7% annual interest compounded monthly,
how much will I have after 30 years?
    """)

    await asyncio.sleep(2)

    # Example 2: Loan Payment
    await calculate_financial("""
Calculate the monthly payment on a $300,000 mortgage at 6.5%
annual interest for 30 years.
    """)

    await asyncio.sleep(2)

    # Example 3: Retirement Planning
    await calculate_financial("""
I'm 30 years old and want to retire at 65 with $2 million.
If I can earn 8% annually, how much do I need to save monthly?
    """)

    await asyncio.sleep(2)

    # Example 4: Investment Comparison
    await calculate_financial("""
Compare two investments:
A) $50,000 initial investment, 6% annual return for 20 years
```

```
B) $30,000 initial + $200/month, 8% annual return for 20 years

Which is better?
    """)

    await asyncio.sleep(2)

    # Example 5: Break-even Analysis
    await calculate_financial("""
A business has fixed costs of $50,000/month and variable costs
of $25 per unit. If they sell units for $75 each, what's the
break-even point?
    """)

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

Expected Output

```
======================================================================
QUERY: If I invest $10,000 at 7% annual interest compounded monthly,
how much will I have after 30 years?
======================================================================

💰 CALCULATION:

Let me calculate the compound interest:

[Code:]
# Compound Interest Formula: A = P(1 + r/n)^(nt)
# Where:
# P = principal amount ($10,000)
# r = annual interest rate (0.07)
# n = number of times interest compounds per year (12)
# t = number of years (30)

principal = 10000
rate = 0.07
compounds_per_year = 12
years = 30

# Calculate future value
future_value = principal * (1 + rate/compounds_per_year) ** (compounds_per_yea

# Calculate total interest earned
interest_earned = future_value - principal

print(f"Initial Investment: ${principal:,.2f}")
print(f"Future Value: ${future_value:,.2f}")
print(f"Interest Earned: ${interest_earned:,.2f}")
print(f"Multiple: {future_value/principal:.2f}x")

[Result:]
Initial Investment: $10,000.00
Future Value: $81,402.45
Interest Earned: $71,402.45
Multiple: 8.14x

**Results:**
- Your investment will grow to **$81,402.45**
- You'll earn **$71,402.45** in interest
- Your money will multiply **8.14 times** over 30 years

**Key Insight:** The power of compound interest! By compounding monthly
instead of annually, you gain an extra ~$3,000 compared to annual compounding.
```

```
======================================================================

======================================================================
QUERY: Calculate the monthly payment on a $300,000 mortgage at 6.5%
annual interest for 30 years.
======================================================================

💰 CALCULATION:

Let me calculate the mortgage payment:

[Code:]
# Mortgage Payment Formula: M = P[r(1+r)^n]/[(1+r)^n-1]
# Where:
# P = loan principal ($300,000)
# r = monthly interest rate (annual rate / 12)
# n = number of payments (years * 12)

principal = 300000
annual_rate = 0.065
monthly_rate = annual_rate / 12
num_payments = 30 * 12

# Calculate monthly payment
numerator = monthly_rate * (1 + monthly_rate) ** num_payments
denominator = (1 + monthly_rate) ** num_payments - 1
monthly_payment = principal * (numerator / denominator)

# Calculate total paid and interest
total_paid = monthly_payment * num_payments
total_interest = total_paid - principal

print(f"Loan Amount: ${principal:,.2f}")
print(f"Monthly Payment: ${monthly_payment:,.2f}")
print(f"Total Paid: ${total_paid:,.2f}")
print(f"Total Interest: ${total_interest:,.2f}")
print(f"Interest as % of Principal: {(total_interest/principal)*100:.1f}%")

[Result:]
Loan Amount: $300,000.00
Monthly Payment: $1,896.20
Total Paid: $682,632.00
Total Interest: $382,632.00
Interest as % of Principal: 127.5%

**Results:**
```

```
- Monthly payment: **$1,896.20**
- Total amount paid over 30 years: **$682,632**
- Total interest paid: **$382,632**

**Important Note:** You'll pay 127.5% of the original loan amount in interest
alone! Consider making extra principal payments to reduce this significantly.

======================================================================
```

4. Advanced Code Execution Patterns

Pattern 1: Visualization Code Generation (For Local Execution)

```python
viz_agent = Agent(
    model='gemini-2.0-flash',
    name='data_viz',
    instruction="""
Generate Python code for data visualizations using matplotlib.
Show the code that would create the visualization.
⚠️ IMPORTANT: This code is for users to run LOCALLY - matplotlib
cannot be executed within ADK's sandboxed environment.
    """,
    code_executor=BuiltInCodeExecutor()
)

result = runner.run(
    "Generate code to create a bar chart showing sales by quarter: " +
    "Q1=50k, Q2=65k, Q3=72k, Q4=80k",
    agent=viz_agent
)
```

⚠️ **CRITICAL LIMITATION**: The code below cannot be executed within ADK's code execution environment. This is **sample output** showing what the agent would generate for users to run on their own systems with matplotlib installed locally.

Output (code for user to run locally - NOT executable in ADK):

```python
import matplotlib.pyplot as plt

quarters = ['Q1', 'Q2', 'Q3', 'Q4']
sales = [50000, 65000, 72000, 80000]

plt.figure(figsize=(10, 6))
plt.bar(quarters, sales, color='steelblue')
plt.title('Quarterly Sales Performance', fontsize=16, fontweight='bold')
plt.xlabel('Quarter', fontsize=12)
plt.ylabel('Sales ($)', fontsize=12)
plt.ylim(0, max(sales) * 1.1)

# Add value labels on bars
for i, v in enumerate(sales):
    plt.text(i, v + 1000, f'${v:,}', ha='center', va='bottom')

plt.grid(axis='y', alpha=0.3)
plt.tight_layout()
plt.show()
```

What ADK Code Execution CANNOT Do:

- ❌ Generate actual graphics or charts
- ❌ Use matplotlib, seaborn, plotly, or any visualization libraries
- ❌ Display images or plots
- ❌ Save chart files

What ADK Code Execution CAN Do:

- ✔️ Generate matplotlib code as text for local execution
- ✔️ Perform all mathematical calculations
- ✔️ Create text-based data representations
- ✔️ Generate ASCII art or simple text charts
- ✔️ Analyze data and provide insights

Pattern 2: Scientific Calculations

```
science_agent = Agent(
    model='gemini-2.0-flash',
    name='scientist',
    instruction='Perform scientific calculations and simulations using Python.
    code_executor=BuiltInCodeExecutor()
)

result = runner.run(
    """
Calculate the orbital period of a satellite at 400km altitude above Earth.
Use: G = 6.674×10^-11 N·m²/kg², Earth mass = 5.972×10^24 kg,
Earth radius = 6371 km
    """,
    agent=science_agent
)
```

Pattern 3: Statistical Analysis

```
stats_agent = Agent(
    model='gemini-2.0-flash',
    name='statistician',
    instruction='Perform statistical analysis including hypothesis ' +
        'testing and confidence intervals.',
    code_executor=BuiltInCodeExecutor()
)

result = runner.run(
    """
Given sample data [23, 25, 28, 30, 29, 27, 26, 24, 31, 28]:
1. Calculate mean, median, standard deviation
2. Construct 95% confidence interval for the mean
3. Test if mean is significantly different from 25 (α=0.05)
    """,
    agent=stats_agent
)
```

Pattern 4: Algorithm Optimization

```
optimizer_agent = Agent(
    model='gemini-2.0-flash',
    name='optimizer',
    instruction='Implement and compare algorithm efficiency using Python.',
    code_executor=BuiltInCodeExecutor()
)

result = runner.run(
    """
Compare bubble sort vs quicksort performance on a list of 1000 random numbers.
Measure execution time and number of comparisons for each.
    """,
    agent=optimizer_agent
)
```

5. Combining Code Execution with Tools

You can combine code execution with other tools for powerful agents:

```python
from google.adk.agents import Agent, Runner
from google.adk.code_executors import BuiltInCodeExecutor
from google.adk.tools import FunctionTool, GoogleSearchAgentTool

def get_stock_data(symbol: str) -> dict:
    """Simulated stock data fetcher."""
    # In production, call real financial API
    return {
        'symbol': symbol,
        'prices': [150, 152, 148, 155, 153, 157, 160],
        'volume': [1000000, 1100000, 950000, 1200000, 1050000, 1300000, 125000
    }

# Agent with code execution AND custom tools
hybrid_agent = Agent(
    model='gemini-2.0-flash',
    name='financial_analyst',
    instruction="""
You are a financial analyst with:
1. get_stock_data tool to fetch market data
2. Code execution to analyze the data
3. Web search to find company news

Use all capabilities to provide comprehensive analysis.
    """,
    code_executor=BuiltInCodeExecutor(),
    tools=[
        FunctionTool(get_stock_data),
        GoogleSearchAgentTool()
    ]
)

runner = Runner()
result = runner.run(
    "Analyze AAPL stock performance and calculate volatility",
    agent=hybrid_agent
)
```

6. Best Practices

✔️ DO: Use Code Execution for Precision

```python
# ✔️ Good - Use code for exact calculations
agent = Agent(
    model='gemini-2.0-flash',
    instruction='Use Python code for all mathematical calculations.',
    code_executor=BuiltInCodeExecutor()
)

# ❌ Bad - Let model approximate
agent = Agent(
    model='gemini-2.0-flash',
    instruction='Approximate calculations in your head.'
)
```

✔️ DO: Set Low Temperature for Accuracy

```python
# ✔️ Good - Low temperature for code generation
agent = Agent(
    model='gemini-2.0-flash',
    code_executor=BuiltInCodeExecutor(),
    generate_content_config=types.GenerateContentConfig(
        temperature=0.1  # More deterministic code
    )
)

# ❌ Bad - High temperature can produce invalid code
agent = Agent(
    model='gemini-2.0-flash',
    code_executor=BuiltInCodeExecutor(),
    generate_content_config=types.GenerateContentConfig(
        temperature=0.9  # Too creative for code
    )
)
```

✔️ DO: Provide Clear Instructions

```
# ✔️ Good - Clear guidance
agent = Agent(
    model='gemini-2.0-flash',
    instruction="""
For calculations:
1. Always write Python code
2. Show the code you're executing
3. Explain the logic
4. Display results clearly
5. Provide interpretation
    """,
    code_executor=BuiltInCodeExecutor()
)

# ❌ Bad - Vague
agent = Agent(
    model='gemini-2.0-flash',
    instruction="Do calculations",
    code_executor=BuiltInCodeExecutor()
)
```

✔️ DO: Handle Edge Cases

```
# ✔️ Good - Instruct about error handling
agent = Agent(
    model='gemini-2.0-flash',
    instruction="""
When writing code:
1. Check for division by zero
2. Validate input ranges
3. Handle edge cases (empty lists, negatives, etc.)
4. Include try-except for errors
5. Provide meaningful error messages
    """,
    code_executor=BuiltInCodeExecutor()
)
```

✔️ DO: Verify Results

```
# ✔️ Good - Ask agent to verify
agent = Agent(
    model='gemini-2.0-flash',
    instruction="""
After executing code:
1. Check if result makes sense
2. Verify with alternative method if possible
3. Note any assumptions made
4. Warn about limitations
    """,
    code_executor=BuiltInCodeExecutor()
)
```

7. Troubleshooting

Error: "Code execution requires Gemini 2.0+"

Problem: Using code executor with wrong model

Solution:

```
# ❌ Wrong model version
agent = Agent(
    model='gemini-1.5-flash',
    code_executor=BuiltInCodeExecutor()  # Error
)

# ✔️ Use Gemini 2.0+
agent = Agent(
    model='gemini-2.0-flash',
    code_executor=BuiltInCodeExecutor()
)
```

Issue: "Code not executing"

Problem: Model not using code execution feature

Solutions:

1. **Make query require computation**:

```
# ❌ Model might not execute code
result = runner.run("What's 2+2?", agent=agent)

# ✔️ Complex calculation triggers code execution
result = runner.run("Calculate the standard deviation of " +
    "[1,2,3,4,5,6,7,8,9,10]", agent=agent)
```

1. **Explicit instruction**:

```
agent = Agent(
    model='gemini-2.0-flash',
    instruction='ALWAYS write and execute Python code for calculations. Never
    code_executor=BuiltInCodeExecutor()
)
```

Issue: "Code execution errors"

Problem: Generated code has bugs

Solutions:

1. **Lower temperature**:

```
agent = Agent(
    model='gemini-2.0-flash',
    code_executor=BuiltInCodeExecutor(),
    generate_content_config=types.GenerateContentConfig(
        temperature=0.0  # Most deterministic
    )
)
```

1. **Add error handling instruction**:

```
agent = Agent(
    model='gemini-2.0-flash',
    instruction="""
When writing code:
- Test with simple cases first
- Use try-except blocks
- Validate inputs
- Check for edge cases
    """,
    code_executor=BuiltInCodeExecutor()
)
```

Issue: "Slow response time"

Problem: Code execution adds latency

Solutions:

1. **Use streaming**:

```
from google.adk.agents import RunConfig, StreamingMode

run_config = RunConfig(streaming_mode=StreamingMode.SSE)

async for event in runner.run_async(query, agent=agent, run_config=run_config)
    print(event.content.parts[0].text, end='', flush=True)
```

1. **Optimize code complexity**:

```
agent = Agent(
    model='gemini-2.0-flash',
    instruction='Write efficient code. Avoid unnecessary loops or complex oper
    code_executor=BuiltInCodeExecutor()
)
```

8. Testing Code Execution Agents

Unit Tests

```python
import pytest
from google.adk.agents import Agent, Runner
from google.adk.code_executors import BuiltInCodeExecutor

@pytest.mark.asyncio
async def test_code_execution_accuracy():
    """Test that code execution provides accurate results."""

    agent = Agent(
        model='gemini-2.0-flash',
        code_executor=BuiltInCodeExecutor()
    )

    runner = Runner()
    result = await runner.run_async(
        "Calculate factorial of 10",
        agent=agent
    )

    # Factorial of 10 = 3,628,800
    assert '3628800' in result.content.parts[0].text

@pytest.mark.asyncio
async def test_statistical_calculation():
    """Test statistical calculations."""

    agent = Agent(
        model='gemini-2.0-flash',
        instruction='Calculate exact statistics using Python.',
        code_executor=BuiltInCodeExecutor(),
        generate_content_config=types.GenerateContentConfig(temperature=0.1)
    )

    runner = Runner()
    result = await runner.run_async(
        "Calculate the mean of [10, 20, 30, 40, 50]",
        agent=agent
    )

    # Mean should be 30
    text = result.content.parts[0].text
    assert '30' in text or 'thirty' in text.lower()

@pytest.mark.asyncio
async def test_complex_calculation():
    """Test complex mathematical calculation."""
    agent = Agent(
        model='gemini-2.0-flash',
        code_executor=BuiltInCodeExecutor()
    )

    runner = Runner()
    result = await runner.run_async(
        "Calculate compound interest: $1000 principal, 5% annual rate, " +
        "10 years, monthly compounding",
        agent=agent
    )

    text = result.content.parts[0].text
    # Should be around $1647
    assert '1647' in text or '1,647' in text

@pytest.mark.asyncio
async def test_algorithm_implementation():
    """Test that agent can implement algorithms."""

    agent = Agent(
        model='gemini-2.0-flash',
        instruction='Implement algorithms using Python code.',
        code_executor=BuiltInCodeExecutor()
    )

    runner = Runner()
    result = await runner.run_async(
        "Implement a function to check if a number is prime, then test it with
        agent=agent
    )

    text = result.content.parts[0].text.lower()
    # 17 is prime
    assert 'true' in text or 'prime' in text
```

9. Security Considerations

Code Execution Security

Important: Code executes in Google's model environment, not locally. This provides security benefits:

✓ **Isolated Environment**: Code runs in sandboxed model environment
✓ **No Local Access**: Cannot access your local file system
✓ **No Network Access**: Cannot make external network calls
✓ **Limited Resources**: Resource-constrained execution
✓ **Automatic Cleanup**: No persistent state between executions

What Code CAN Do:

- Mathematical calculations
- Data processing (lists, dicts, arrays)
- Algorithm implementation
- String manipulation
- Statistical analysis

What Code CANNOT Do:

- Access local files
- Make network requests
- Install packages
- Execute shell commands
- Access environment variables
- Persist data between executions

Best Practices for Production

```
# ✔️ Good - Clear boundaries
agent = Agent(
    model='gemini-2.0-flash',
    instruction="""
You can use Python for:
- Calculations
- Data analysis
- Algorithm implementation

You CANNOT:
- Access files
- Make network requests
- Execute system commands
    """,
    code_executor=BuiltInCodeExecutor()
)
```

Summary

You've mastered code execution for AI agents:

Key Takeaways:

- ✔️ `BuiltInCodeExecutor` enables Python code generation and execution
- ✔️ Code runs **inside model environment** (Google's infrastructure)
- ✔️ Requires **Gemini 2.0+** models
- ✔️ Perfect for: calculations, data analysis, algorithms, statistics
- ✔️ More accurate than model approximations
- ✔️ Secure - isolated sandbox execution
- ✔️ Can combine with other tools (search, custom functions)
- ✔️ Best with low temperature (0.0-0.1) for accuracy

Production Checklist:

- [ ] Using Gemini 2.0+ model
- [ ] Low temperature set (0.0-0.2)

- [ ] Clear instructions for when to use code
- [ ] Error handling instructions included
- [ ] Testing with various calculation types
- [ ] Streaming enabled for better UX
- [ ] Verification step in instructions
- [ ] Edge case handling specified

Next Steps:

- **Tutorial 14**: Implement Streaming (SSE) for real-time responses
- **Tutorial 15**: Explore Live API for voice and bidirectional streaming
- **Tutorial 16**: Learn MCP Integration for extended tool ecosystem

Resources:

- ADK Code Execution Docs (https://ai.google.dev/gemini-api/docs/code-execution/)
- Gemini 2.0 Code Execution (https://cloud.google.com/vertex-ai/generative-ai/docs/model-reference/gemini)
- Python Standard Library (https://docs.python.org/3/library/)

🎉 **Tutorial 13 Complete!** You now know how to build agents that can write and execute code for accurate calculations. Continue to Tutorial 14 to learn about streaming responses.

Generated on 2025-10-19 17:56:44 from 13_code_execution.md

Source: Google ADK Training Hub