Tutorial 20: YAML Configuration - Declarative Agent Setup

Difficulty: intermediate

Reading Time: 45 minutes

Tags: intermediate, yaml, configuration, declarative, setup

Description: Configure agents using YAML files for declarative setup, easier maintenance, and configuration management across environments.

Tutorial 20: Agent Configuration with YAML

Goal: Master declarative agent configuration using YAML files to define agents, tools, and behaviors without writing Python code, enabling rapid prototyping and configuration management.

Prerequisites:

- Tutorial 01 (Hello World Agent)
- Tutorial 02 (Function Tools)
- Tutorial 06 (Multi-Agent Systems)
- Basic understanding of YAML syntax

What You'll Learn:

- Creating agent configurations with `root_agent.yaml`
- Understanding `AgentConfig` and `LlmAgentConfig` schemas
- Configuring tools, models, and instructions in YAML
- Multi-agent systems in configuration files

- When to use YAML vs Python code
- Loading and validating configurations
- Best practices for config management

Time to Complete: 45 minutes

Why YAML Configuration Matters

Problem: Writing Python code for every agent configuration requires development expertise and makes rapid iteration difficult.

Solution: **YAML configuration** enables declarative agent definitions that can be edited without code changes.

Benefits:

- 🚀 **Rapid Prototyping**: Change configurations without coding
- 📝 **Readable**: Human-friendly format
- [FLOW] **Version Control**: Easy to track config changes
- 🎯 **Separation**: Configuration separate from implementation
- 👥 **Accessibility**: Non-developers can modify agents
- 🔧 **Reusable**: Share configurations across projects

Use Cases:

- Quick agent prototyping
- Configuration-driven deployments
- Multi-environment setups (dev, staging, prod)
- Agent marketplace/templates
- Non-technical team member modifications

Status: YAML configuration is marked as `@experimental` in ADK. API may change.

:::info API Verification

Source Verified: Official ADK source code (version 1.16.0+)

Correct API: `config_agent_utils.from_config(config_path)`

Common Mistake: Using `AgentConfig.from_yaml_file()` - this method **does not exist**. Instead, use `config_agent_utils.from_config()` which loads the YAML file and returns a ready-to-use agent instance.

Verification Date: October 2025

:::

---

1. YAML Configuration Basics

What is root_agent.yaml?

`root_agent.yaml` is the main configuration file that defines an agent and its sub-agents declaratively.

Location: Place in project root or specify path explicitly.

Basic Structure:

```
root_agent.yaml
├── name (required)
├── model (required)
├── description (optional)
├── instruction (optional)
├── generate_content_config (optional)
|   ├── temperature
|   ├── max_output_tokens
|   ├── top_p
|   └── top_k
├── tools (optional)
|   └── [tool_name, ...]
└── sub_agents (optional)
    └── [agent_config, ...]
```

```yaml
# root_agent.yaml

name: my_agent
model: gemini-2.0-flash
description: A helpful agent
instruction: |
  You are a helpful assistant that answers questions
  accurately and concisely.

generate_content_config:
  temperature: 0.7
  max_output_tokens: 1024

tools:
  - type: function
    name: get_weather
    description: Get current weather for a location

sub_agents:
  - name: specialized_agent
    model: gemini-2.0-flash
    description: Specialized agent for specific tasks
```

Creating Configuration Project

```bash
# Create new config-based project
adk create --type=config my_agent_config

# Directory structure created:
# my_agent_config/
#   root_agent.yaml      # Agent configuration
#   tools/               # Custom tool implementations
#   README.md
```

2. AgentConfig Schema

Core Fields

Source: `google/adk/agents/agent_config.py`

```yaml
# Required fields
name: agent_name # Unique identifier
model: gemini-2.0-flash # Model to use

# Optional fields
description: "Agent purpose" # Brief description
instruction: | # System instruction
  Multi-line instruction
  for the agent

# Content generation config
generate_content_config:
  temperature: 0.7 # 0.0-1.0 (creativity)
  max_output_tokens: 2048 # Max response length
  top_p: 0.95 # Nucleus sampling
  top_k: 40 # Top-k sampling

# Tools configuration
tools:
  - type: function
    name: tool_name
    # ... tool config

# Sub-agents
sub_agents:
  - name: sub_agent_1
    # ... agent config
```

Model Options

```yaml
# Gemini 2.0 models (recommended)
model: gemini-2.0-flash         # Fast, efficient
model: gemini-2.0-flash-thinking  # With thinking capability

# Gemini 1.5 models
model: gemini-1.5-flash         # Fast, cost-effective
model: gemini-1.5-pro           # High quality

# Live API models
model: gemini-2.0-flash-live-preview-04-09  # Vertex AI Live
model: gemini-live-2.5-flash-preview        # AI Studio Live
```

3. Real-World Example: Customer Support System

Let's build a complete customer support system using YAML configuration.

Complete Configuration

```yaml
# root_agent.yaml

name: customer_support
model: gemini-2.0-flash
description: Customer support agent with various tools

instruction: |
  You are a customer support agent. Your role is to:

  1. Understand customer inquiries
  2. Use available tools to provide accurate information
  3. Provide comprehensive solutions

  Available tools:
  - check_customer_status: Check if customer is premium member
  - log_interaction: Log customer interaction for records
  - get_order_status: Get status of an order by ID
  - track_shipment: Get shipment tracking information
  - cancel_order: Cancel an order (requires authorization)
  - search_knowledge_base: Search technical documentation
  - run_diagnostic: Run diagnostic tests
  - create_ticket: Create support ticket for escalation
  - get_billing_history: Retrieve billing history
  - process_refund: Process refund (requires approval for amounts > $100)
  - update_payment_method: Update stored payment method

  Guidelines:
  - Always be polite and professional
  - Provide specific information when available
  - Escalate complex issues when necessary

generate_content_config:
  temperature: 0.5
  max_output_tokens: 2048

tools:
  - name: customer_support.tools.check_customer_status
  - name: customer_support.tools.log_interaction
  - name: customer_support.tools.get_order_status
  - name: customer_support.tools.track_shipment
  - name: customer_support.tools.cancel_order
  - name: customer_support.tools.search_knowledge_base
  - name: customer_support.tools.run_diagnostic
  - name: customer_support.tools.create_ticket
  - name: customer_support.tools.get_billing_history
```

```yaml
    - name: customer_support.tools.process_refund
    - name: customer_support.tools.update_payment_method
```

Tool Implementations

```python
# tools/customer_tools.py

"""
Tool implementations for customer support system.
These functions are referenced by name in root_agent.yaml.
"""

def check_customer_status(customer_id: str) -> Dict[str, Any]:
    """
    Check if customer is premium member.

    Args:
        customer_id: Customer identifier

    Returns:
        Dict with status, report, and customer tier information
    """
    # Simulated lookup - in production, would query database
    premium_customers = ['CUST-001', 'CUST-003', 'CUST-005']

    is_premium = customer_id in premium_customers
    tier = 'premium' if is_premium else 'standard'

    return {
        'status': 'success',
        'report': f'Customer {customer_id} is {tier} member',
        'data': {
            'customer_id': customer_id,
            'tier': tier,
            'is_premium': is_premium
        }
    }

def log_interaction(customer_id: str, interaction_type: str, summary: str) ->
    """
    Log customer interaction for records.

    Args:
        customer_id: Customer identifier
        interaction_type: Type of interaction (inquiry, complaint, etc.)
        summary: Brief summary of the interaction

    Returns:
        Dict with status and confirmation
    """
    # In production, would log to database or CRM system
    print(f"[LOG] {customer_id} - {interaction_type}: {summary}")

    return {
        'status': 'success',
        'report': 'Interaction logged successfully',
        'data': {
            'customer_id': customer_id,
            'interaction_type': interaction_type,
            'summary': summary,
            'timestamp': '2025-10-13T10:00:00Z'  # Would be actual timestamp
        }
    }

def get_order_status(order_id: str) -> Dict[str, Any]:
    """
    Get status of an order by ID.

    Args:
        order_id: Order identifier

    Returns:
        Dict with order status information
    """
    # Simulated order lookup - in production, would query order database
    orders = {
        'ORD-001': {'status': 'shipped', 'date': '2025-10-08'},
        'ORD-002': {'status': 'processing', 'date': '2025-10-10'},
        'ORD-003': {'status': 'delivered', 'date': '2025-10-07'},
        'ORD-004': {'status': 'cancelled', 'date': '2025-10-09'}
    }

    order = orders.get(order_id)
    if not order:
        return {
            'status': 'error',
            'error': f'Order {order_id} not found',
            'report': f'No order found with ID {order_id}'
        }

    return {
        'status': 'success',
        'report': f'Order {order_id} status: {order["status"]}',
        'data': {
            'order_id': order_id,
            'status': order['status'],
            'order_date': order['date']
        }
    }

def track_shipment(order_id: str) -> Dict[str, Any]:
    """
    Get shipment tracking information.

    Args:
        order_id: Order identifier

    Returns:
        Dict with tracking information
    """
    # Simulated tracking lookup - in production, would query shipping API
    tracking = {
        'ORD-001': {
            'carrier': 'UPS',
            'tracking_number': '1Z999AA10123456784',
            'estimated_delivery': '2025-10-10',
            'status': 'In transit'
        },
        'ORD-003': {
            'carrier': 'FedEx',
            'tracking_number': '7898765432109',
            'estimated_delivery': 'Delivered on 2025-10-07',
            'status': 'Delivered'
        }
    }

    info = tracking.get(order_id)
    if not info:
        return {
            'status': 'error',
            'error': f'No tracking available for order {order_id}',
            'report': f'No tracking information found for {order_id}'
        }

    return {
        'status': 'success',
        'report': f'Tracking: {info["carrier"]} {info["tracking_number"]}, ETA
        'data': {
            'order_id': order_id,
            'carrier': info['carrier'],
            'tracking_number': info['tracking_number'],
            'estimated_delivery': info['estimated_delivery'],
            'status': info['status']
        }
    }
```

```
def cancel_order(order_id: str, reason: str) -> Dict[str, Any]:
    """
    Cancel an order (requires authorization).

    Args:
        order_id: Order identifier
        reason: Reason for cancellation

    Returns:
        Dict with cancellation status
    """
    # Simulated order cancellation - in production, would have authorization c
    cancellable_orders = ['ORD-001', 'ORD-002']  # Only processing/shipped ord

    if order_id not in cancellable_orders:
        return {
            'status': 'error',
            'error': f'Order {order_id} cannot be cancelled',
            'report': f'Order {order_id} is not eligible for cancellation'
        }

    return {
        'status': 'success',
        'report': f'Order {order_id} cancelled. Reason: {reason}',
        'data': {
            'order_id': order_id,
            'reason': reason,
            'refund_status': 'pending',
            'cancelled_at': '2025-10-13T10:00:00Z'
        }
    }

def search_knowledge_base(query: str) -> Dict[str, Any]:
    """
    Search technical documentation.

    Args:
        query: Search query

    Returns:
        Dict with relevant documentation
    """
    # Simulated knowledge base search - in production, would query documentati
    kb = {
        'login': 'To reset password, go to Settings > Security > Reset Passwor
        'connection': 'Check internet connection and restart the app',
        'error': 'Clear app cache: Settings > Apps > Clear Cache',
        'update': 'Go to Settings > Updates > Check for Updates',
        'sync': 'Ensure device is connected and try Settings > Sync > Sync Now
    }

    query_lower = query.lower()
    results = []

    for key, value in kb.items():
        if key in query_lower:
            results.append({
                'topic': key,
                'solution': value
            })

    if not results:
        return {
            'status': 'success',
            'report': 'No matching article found',
            'data': {
                'query': query,
                'results': [],
                'suggestion': 'Try searching for: login, connection, error, up
            }
        }

    return {
        'status': 'success',
        'report': f'Found {len(results)} relevant article(s)',
        'data': {
            'query': query,
            'results': results
        }
    }

def run_diagnostic(issue_type: str) -> Dict[str, Any]:
    """
    Run diagnostic tests.

    Args:
        issue_type: Type of issue to diagnose

    Returns:
        Dict with diagnostic results
    """
    # Simulated diagnostic - in production, would run actual diagnostic tests
    diagnostics = {
        'connection': {
            'tests': ['Network connectivity', 'Server response', 'DNS resoluti
            'result': 'All systems operational',
            'recommendation': 'Clear cache and restart'
        },
        'performance': {
            'tests': ['Memory usage', 'CPU usage', 'Disk space'],
            'result': 'Performance within normal range',
            'recommendation': 'Close unused applications'
        },
        'login': {
            'tests': ['Authentication service', 'Session management', 'Passwor
            'result': 'Authentication systems operational',
            'recommendation': 'Check password and try again'
        }
    }

    diagnostic = diagnostics.get(issue_type.lower())
    if not diagnostic:
        return {
            'status': 'error',
            'error': f'Unknown issue type: {issue_type}',
            'report': f'No diagnostic available for {issue_type}'
        }

    return {
        'status': 'success',
        'report': f'Diagnostic for {issue_type}: {diagnostic["result"]}. Sugge
        'data': {
            'issue_type': issue_type,
            'tests_run': diagnostic['tests'],
            'result': diagnostic['result'],
            'recommendation': diagnostic['recommendation']
        }
    }

def create_ticket(customer_id: str, issue: str, priority: str) -> Dict[str, An
    """
    Create support ticket for escalation.

    Args:
        customer_id: Customer identifier
        issue: Description of the issue
        priority: Priority level (low, medium, high, urgent)

    Returns:
        Dict with ticket information
    """
    # Simulated ticket creation - in production, would create in ticketing sys
    import random
    ticket_id = f"TKT-{random.randint(1000, 9999):04d}"

    valid_priorities = ['low', 'medium', 'high', 'urgent']
    if priority.lower() not in valid_priorities:
        priority = 'medium'  # Default to medium

    return {
        'status': 'success',
        'report': f'Support ticket {ticket_id} created with {priority} priorit
        'data': {
            'ticket_id': ticket_id,
            'customer_id': customer_id,
            'issue': issue,
            'priority': priority,
            'status': 'open',
            'created_at': '2025-10-13T10:00:00Z',
            'estimated_response': '2 hours' if priority in ['high', 'urgent']
        }
    }

def get_billing_history(customer_id: str) -> Dict[str, Any]:
    """
    Retrieve billing history.

    Args:
        customer_id: Customer identifier

    Returns:
        Dict with billing history
    """
    # Simulated billing lookup - in production, would query billing database
    billing_history = {
        'CUST-001': [
            {'date': '2025-09-01', 'amount': 49.99, 'description': 'Monthly su
            {'date': '2025-08-01', 'amount': 49.99, 'description': 'Monthly su
            {'date': '2025-07-15', 'amount': 29.99, 'description': 'One-time p
        ],
        'CUST-002': [
            {'date': '2025-09-15', 'amount': 19.99, 'description': 'Basic plan
            {'date': '2025-08-15', 'amount': 19.99, 'description': 'Basic plan
        ]
    }

    history = billing_history.get(customer_id, [])
    if not history:
        return {
            'status': 'error',
            'error': f'No billing history found for {customer_id}',
            'report': f'No billing records found for customer {customer_id}'
        }

    total = sum(item['amount'] for item in history)

    return {
        'status': 'success',
        'report': f'Found {len(history)} billing records for {customer_id}',
        'data': {
            'customer_id': customer_id,
            'transactions': history,
            'total_amount': total,
            'currency': 'USD'
        }
    }

def process_refund(order_id: str, amount: float) -> Dict[str, Any]:
    """
    Process refund (requires approval for amounts > $100).

    Args:
        order_id: Order identifier
        amount: Refund amount

    Returns:
        Dict with refund status
    """
    if amount > 100:
        return {
            'status': 'error',
            'error': 'REQUIRES_APPROVAL',
            'report': f'Refund of ${amount} for {order_id} needs manager appro
            'data': {
                'order_id': order_id,
                'amount': amount,
                'status': 'pending_approval',
                'approval_required': True
            }
        }

    return {
        'status': 'success',
        'report': f'Refund of ${amount} approved for {order_id}. Funds will ap
        'data': {
            'order_id': order_id,
            'amount': amount,
            'status': 'approved',
            'processing_time': '3-5 business days',
            'refund_id': f'REF-{order_id}-{amount:.0f}'
        }
    }

def update_payment_method(customer_id: str, payment_type: str) -> Dict[str, An
    """
    Update stored payment method.

    Args:
        customer_id: Customer identifier
        payment_type: New payment method type

    Returns:
        Dict with update confirmation
    """
    # Simulated payment method update - in production, would update payment sy
    valid_types = ['credit_card', 'debit_card', 'paypal', 'bank_transfer']

    if payment_type.lower() not in valid_types:
        return {
            'status': 'error',
            'error': f'Invalid payment type: {payment_type}',
            'report': f'Payment type must be one of: {", ".join(valid_types)}'
        }

    return {
        'status': 'success',
        'report': f'Payment method for {customer_id} updated to {payment_type}
        'data': {
            'customer_id': customer_id,
            'payment_type': payment_type,
            'updated_at': '2025-10-13T10:00:00Z',
            'verification_required': True,
            'status': 'pending_verification'
        }
    }
```

Loading and Running Configuration

Process Flow:

```
root_agent.yaml ──► config_agent_utils.from_config() ──► Agent Instance
                       ├── Validate YAML syntax
                       ├── Resolve tool functions
                       ├── Create agent with config
                       └── Return ready-to-use agent
```

```python
# run_agent.py

"""
Load and run agent from YAML configuration.
"""

import asyncio
import os
from google.adk.agents import Runner, Session
from google.adk.agents import config_agent_utils

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

async def main():
    """Load configuration and run agent."""

    # Load agent from YAML configuration
    agent = config_agent_utils.from_config('root_agent.yaml')

    # Create runner and session
    runner = Runner()
    session = Session()

    # Test queries
    queries = [
        "I'm customer CUST-001 and I want to check my order ORD-001",
        "I need help with a login error",
        "I'd like a refund of $75 for order ORD-002"
    ]

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

        result = await runner.run_async(
            query,
            agent=agent,
            session=session
        )

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

        await asyncio.sleep(2)

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

Expected Output

```
======================================================================
QUERY: Check the status of customer CUST-001
======================================================================

RESPONSE:
Hello! I can help you check the customer status. Let me look that up for you.

Customer CUST-001 is premium member

Is there anything else I can help you with?

======================================================================

======================================================================
QUERY: What's the status of order ORD-001?
======================================================================

RESPONSE:
I'd be happy to check the status of your order. Let me look that up.

Order ORD-001 status: shipped

If you need tracking information or have any other questions about this order,

======================================================================

======================================================================
QUERY: Can you track shipment for order ORD-001?
======================================================================

RESPONSE:
I'll help you track that shipment. Let me get the tracking details.

Tracking: UPS 1Z999AA10123456784, ETA: 2025-10-10

Your package is currently in transit and expected to arrive by October 10th, 2

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

4. YAML vs Python: When to Use Each

Decision Flow: YAML or Python?

```
Need to configure an agent?
├── Is this for rapid prototyping/testing? ──► YAML
├── Do non-technical team members need to edit? ──► YAML
├── Need version control for configurations? ──► YAML
├── Require multi-environment configs? ──► YAML
├── Need complex conditional logic? ──► PYTHON
├── Require dynamic tool selection? ──► PYTHON
├── Need custom components/callbacks? ──► PYTHON
├── Building advanced patterns (loops)? ──► PYTHON
└── Need IDE support (autocomplete)? ──► PYTHON
```

Use YAML Configuration When:

✓ **Rapid prototyping** - Testing different agent configurations
✓ **Non-technical editors** - Allow team members to modify agents
✓ **Configuration management** - Separate config from code
✓ **Multi-environment** - Dev, staging, prod configurations
✓ **Simple workflows** - Standard agent patterns
✓ **Version control** - Track configuration changes easily

Use Python Code When:

✓ **Complex logic** - Conditional tool selection, dynamic workflows
✓ **Custom components** - Custom planners, executors, callbacks
✓ **Advanced patterns** - Loops, complex state management
✓ **Programmatic generation** - Creating agents dynamically
✓ **Testing** - Unit tests, integration tests
✓ **IDE support** - Type checking, autocomplete, refactoring

Hybrid Approach (Best Practice)

Architecture: Combine YAML declarative config with Python programmatic customization.

```
YAML Base Config ──┐
                   ├──► Agent Instance ──┬──► Runtime
Python Code ───────┘                     │
                                         ├──► Custom Tools
                                         ├──► Dynamic Logic
                                         └──► Runtime Adjustments
```

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

# Load base configuration from YAML
agent = config_agent_utils.from_config('base_agent.yaml')

# Customize programmatically
agent.tools.append(custom_complex_tool)
agent.instruction += "\n\nAdditional dynamic instructions"

# Run with custom logic
if user_is_premium:
    agent.tools.append(premium_tool)

runner.run(query, agent=agent)
```

5. Best Practices

✔️ DO: Use Environment-Specific Configs

Directory Structure:

```
config/
├── dev/
|   ├── root_agent.yaml     # Development config
|   └── secrets.yaml        # Dev secrets
├── staging/
|   ├── root_agent.yaml     # Staging config
|   └── secrets.yaml        # Staging secrets
└── prod/
    ├── root_agent.yaml     # Production config
    └── secrets.yaml        # Prod secrets
```

```yaml
# config/dev/root_agent.yaml
name: support_agent_dev
model: gemini-2.0-flash
generate_content_config:
  temperature: 0.8  # More creative for testing

# config/prod/root_agent.yaml
name: support_agent_prod
model: gemini-2.0-flash
generate_content_config:
  temperature: 0.3  # More consistent for production
```

✔️ DO: Document Configuration

```yaml
# root_agent.yaml

# Customer Support Orchestrator
# Maintainer: support-team@example.com
# Last Updated: 2025-10-08
#
# This agent routes customer inquiries to specialized agents:
# - order_agent: Order management
# - technical_agent: Technical support
# - billing_agent: Payment issues

name: customer_support
model: gemini-2.0-flash

instruction: |
  [Clear instruction here]
```

✔️ DO: Validate Configuration

```
from google.adk.agents import config_agent_utils

def validate_config(yaml_path: str) -> bool:
    """Validate agent configuration."""

    try:
        agent = config_agent_utils.from_config(yaml_path)
        print(f"✔️ Configuration valid: {agent.name}")
        return True

    except Exception as e:
        print(f"❌ Configuration error: {e}")
        return False

# Validate before deployment
validate_config('root_agent.yaml')
```

✔️ DO: Version Control Configuration

```
# .gitignore - Don't commit secrets
config/secrets.yaml
*.env

# Git commit configuration changes
git add root_agent.yaml
git commit -m "Update customer_support agent temperature to 0.5"
```

❌ DON'T: Hardcode Secrets

```
# ❌ Bad - secrets in config
tools:
  - type: api
    api_key: "sk-proj-abc123..."  # NEVER do this

# ✔️ Good - reference environment variables
tools:
  - type: api
    api_key: "${API_KEY}"  # Load from environment
```

6. Advanced Configuration Patterns

Pattern 1: Conditional Sub-Agents

```yaml
# Different sub-agents for different tiers
name: support_agent

sub_agents:
  # Basic support (all tiers)
  - name: faq_agent
    model: gemini-2.0-flash
    description: FAQ and basic questions

  # Premium support only (filter in code)
  - name: premium_support_agent
    model: gemini-2.0-flash
    description: Premium customer support
    # Enable only for premium customers in code
```

Pattern 2: Configuration Inheritance

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

# Load base configuration
specialized_agent = config_agent_utils.from_config('config/base.yaml')

# Create specialized variants
specialized_agent.instruction += "\n\nSpecialized for domain X"
specialized_agent.tools.append(domain_specific_tool)
```

Pattern 3: Dynamic Tool Registration

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

# Load config
agent = config_agent_utils.from_config('root_agent.yaml')

# Add tools dynamically based on user permissions
if user.has_permission('admin'):
    agent.tools.append(FunctionTool(admin_tool))

if user.has_permission('data_export'):
    agent.tools.append(FunctionTool(export_tool))
```

7. Troubleshooting

Issue: "Configuration file not found"

Solutions:

1. **Check file path**:

```python
import os
config_path = 'root_agent.yaml'
print(f"Looking for: {os.path.abspath(config_path)}")
print(f"Exists: {os.path.exists(config_path)}")
```

1. **Specify absolute path**:

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

agent = config_agent_utils.from_config('/full/path/to/root_agent.yaml')
```

Issue: "Invalid YAML syntax"

Solution: Validate YAML syntax:

```
# Install yamllint
pip install yamllint

# Validate configuration
yamllint root_agent.yaml
```

Issue: "Tool function not found"

Solution: Ensure tool functions are importable:

```
# tools/__init__.py
from .customer_tools import (
    check_customer_status,
    log_interaction,
    get_order_status
)

__all__ = [
    'check_customer_status',
    'log_interaction',
    'get_order_status'
]
```

Summary

You've mastered YAML agent configuration:

Key Takeaways:

- ✓ `root_agent.yaml` for declarative agent definitions
- ✓ `config_agent_utils.from_config()` to load configurations
- ✓ YAML for rapid prototyping and configuration management
- ✓ Python code for complex logic and customization
- ✓ Hybrid approach combines best of both
- ✓ Environment-specific configs for dev/staging/prod
- ✓ Version control for configuration tracking

Production Checklist:

- [ ] Configuration files version controlled
- [ ] Secrets loaded from environment variables
- [ ] Configuration validation in CI/CD
- [ ] Environment-specific configs (dev/staging/prod)
- [ ] Documentation in YAML comments
- [ ] Tool functions properly registered
- [ ] Configuration tested before deployment
- [ ] Backup of production configurations

Next Steps:

- **Tutorial 21**: Learn Multimodal & Image Generation
- **Tutorial 22**: Master Model Selection & Optimization
- **Tutorial 23**: Explore Production Deployment

Resources:

- ADK Configuration Documentation (https://google.github.io/adk-docs/configuration/)
- AgentConfig API Reference (https://google.github.io/adk-docs/api/agent-config/)
- YAML Specification (https://yaml.org/spec/)

🎉 **Tutorial 20 Complete!** You now know how to configure agents with YAML. Continue to Tutorial 21 to learn about multimodal capabilities and image generation.
