

# Tutorial 09: Callbacks and Guardrails - Agent Safety and Monitoring

---

**Difficulty:** advanced

**Reading Time:** 1.5 hours

**Tags:** advanced, safety, callbacks, guardrails, monitoring

**Description:** Implement safety guardrails and monitoring callbacks to control agent behavior, prevent harmful outputs, and track agent performance.

## Tutorial 09: Callbacks & Guardrails - Control Flow and Monitoring

---

### Overview

---

Learn how to use **callbacks** to observe, customize, and control agent behavior at specific execution points. This tutorial demonstrates a content moderation system with safety guardrails, logging, and request/response modification.

**What You'll Build:** An intelligent content assistant that:

- **Blocks** inappropriate requests before reaching the LLM (guardrails)
- **Validates** tool arguments before execution
- **Logs** all LLM calls and tool executions (monitoring)
- **Modifies** requests to add safety instructions
- **Filters** responses to remove sensitive information

- **Tracks** usage metrics in session state

**Why It Matters:** Production agents need safety checks, monitoring, and control mechanisms. Callbacks provide these without modifying core agent logic.

---

## Prerequisites

---

- Python 3.9+
  - `google-adk` installed ( `pip install google-adk` )
  - Google API key
  - Completed Tutorials 01, 02, and 08 (agents, tools, state management)
- 

## Core Concepts

---

### | What are Callbacks?

**Callbacks** are functions you define that ADK automatically calls at specific execution points. They enable:

- **Observability:** Logging and monitoring
- **Control:** Blocking or modifying operations
- **Customization:** Adapting behavior dynamically
- **Guardrails:** Enforcing safety policies

### | Callback Types

**Agent Lifecycle** (all agent types):

- `before_agent_callback` : Before agent's main logic starts
- `after_agent_callback` : After agent finishes

**LLM Interaction** (LlmAgent only):

- `before_model_callback` : Before LLM API call
-

- `after_model_callback` : After LLM response received

**Tool Execution** (LlmAgent only):

- `before_tool_callback` : Before tool function runs
- `after_tool_callback` : After tool function completes

## | Control Flow Pattern

**Return** `None` → Proceed normally (allow default behavior)

**Return Object** → Override/skip operation:

- `before_agent_callback` → `Content` : Skip agent execution
- `before_model_callback` → `LlmResponse` : Skip LLM call, use returned response
- `before_tool_callback` → `dict` : Skip tool execution, use returned result
- `after_agent_callback` → `Content` : Replace agent output
- `after_model_callback` → `LlmResponse` : Replace LLM response
- `after_tool_callback` → `dict` : Replace tool result

## Use Case: Content Moderation Assistant

**Scenario:** Build a writing assistant that:

- Blocks requests with profanity or hate speech
- Validates tool arguments (e.g., no negative word counts)
- Logs all LLM calls for audit trail
- Adds safety instructions to every LLM request
- Filters PII (personally identifiable info) from responses
- Tracks usage metrics (LLM calls, tool uses, blocked requests)

**Safety Requirements:**

- ✓ Block inappropriate inputs (before they reach LLM)
- ✓ Validate tool arguments (before execution)
- ✓ Log everything (for compliance/debugging)

- ✓ Filter outputs (remove sensitive data)
- ✓ Track metrics (for monitoring)

---

## Implementation

---

### | Project Structure

```
content_moderator/  
├── __init__.py      # Imports agent  
├── agent.py         # Agent definition with callbacks  
└── .env            # API key
```

### | Complete Code

**content\_moderator/init.py:**

```
from .agent import root_agent  
  
__all__ = ['root_agent']
```

**content\_moderator/agent.py:**

```
"""
```

## Content Moderation Assistant - Demonstrates Callbacks & Guardrails

This agent uses callbacks for:

- Guardrails: Block inappropriate content (before\_model\_callback)
- Validation: Check tool arguments (before\_tool\_callback)
- Logging: Track all operations (multiple callbacks)
- Modification: Add safety instructions (before\_model\_callback)
- Filtering: Remove PII from responses (after\_model\_callback)
- Metrics: Track usage statistics (state management)

```
"""
```

```
from google.adk.agents import Agent, CallbackContext
from google.adk.tools.tool_context import ToolContext
from google.genai import types
from typing import Dict, Any, Optional
import re
import logging
```

```
# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)
```

```
# =====
# BLOCKLIST CONFIGURATION
# =====
```

```
# Simplified blocklist for demonstration
BLOCKED_WORDS = [
    'profanity1', 'profanity2', 'hate-speech', # Replace with real terms
    'offensive-term', 'inappropriate-word'
]
```

```
# PII patterns to filter
PII_PATTERNS = {
    'email': r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b',
    'phone': r'\b\d{3}[-.]?\d{3}[-.]?\d{4}\b',
    'ssn': r'\b\d{3}-\d{2}-\d{4}\b',
    'credit_card': r'\b\d{4}[-\s]?d{4}[-\s]?d{4}[-\s]?d{4}\b'
}
```

```
# =====
# CALLBACK FUNCTIONS
# =====
```

```
def before_agent_callback(callback_context: CallbackContext) -> Optional[types
```

```

"""
Called before agent starts processing a request.

Use Case: Check if agent should even handle this request.

Returns:
    None: Allow agent to proceed
    Content: Skip agent execution, use returned content as response
"""
logger.info(f"[AGENT START] Session: {callback_context.invocation_id}")

# Check if agent is in maintenance mode (app state)
if callback_context.state.get('app:maintenance_mode', False):
    logger.warning("[AGENT BLOCKED] Maintenance mode active")
    return types.Content(
        parts=[types.Part(text="System is currently under maintenance. Please try again later.",
                           role="model")]
    )

# Increment request counter
count = callback_context.state.get('user:request_count', 0)
callback_context.state['user:request_count'] = count + 1

return None # Allow agent to proceed

def after_agent_callback(callback_context: CallbackContext, content: types.Content) -> types.Content:
    """
    Called after agent completes processing.

    Use Case: Post-process or validate final output.

    Returns:
        None: Use agent's original output
        Content: Replace agent's output with this
    """
    logger.info(f"[AGENT COMPLETE] Generated {len(content.parts)} parts")

    # Track successful completions
    callback_context.state['temp:agent_completed'] = True

    # Could add standard disclaimer here
    # return types.Content(
    #     parts=content.parts + [types.Part(text="\n\n[This is AI-generated content. Please use responsibly.]",
    #                                         role="model")]
    # )

    return None # Use original output

```

```

def before_model_callback(
    callback_context: CallbackContext,
    llm_request: types.GenerateContentRequest
) -> Optional[types.GenerateContentResponse]:
    """
    Called before sending request to LLM.

    Use Cases:
    1. Guardrails: Block inappropriate requests
    2. Modification: Add safety instructions
    3. Caching: Return cached responses
    4. Logging: Track LLM usage

    Returns:
        None: Allow LLM call to proceed
        LlmResponse: Skip LLM call, use this response instead
    """
    # Extract user input
    user_text = ""
    for content in llm_request.contents:
        for part in content.parts:
            if part.text:
                user_text += part.text

    logger.info(f"[LLM REQUEST] Length: {len(user_text)} chars")

    # GUARDRAIL: Check for blocked words
    for word in BLOCKED_WORDS:
        if word.lower() in user_text.lower():
            logger.warning(f"[LLM BLOCKED] Found blocked word: {word}")

    # Track blocked requests
    blocked_count = callback_context.state.get('user:blocked_requests')
    callback_context.state['user:blocked_requests'] = blocked_count + 1

    # Return error response (skip LLM call)
    return types.GenerateContentResponse(
        candidates=[
            types.Candidate(
                content=types.Content(
                    parts=[types.Part(
                        text="I cannot process this request as it contains blocked words."
                    )],
                    role="model"
                )
            )
        ]
    )

```

```

    )

    # MODIFICATION: Add safety instruction
    safety_instruction = "\n\nIMPORTANT: Do not generate harmful, biased, or i

    # Modify system instruction
    if llm_request.config and llm_request.config.system_instruction:
        llm_request.config.system_instruction += safety_instruction

    # Track LLM calls
    llm_count = callback_context.state.get('user:llm_calls', 0)
    callback_context.state['user:llm_calls'] = llm_count + 1

    return None # Allow LLM call with modifications

def after_model_callback(
    callback_context: CallbackContext,
    llm_response: types.GenerateContentResponse
) -> Optional[types.GenerateContentResponse]:
    """
    Called after receiving response from LLM.

    Use Cases:
    1. Filtering: Remove PII or sensitive data
    2. Formatting: Standardize output format
    3. Logging: Track response quality

    Returns:
        None: Use original LLM response
        LlmResponse: Replace with modified response
    """

    # Extract response text
    response_text = ""
    if llm_response.candidates:
        for part in llm_response.candidates[0].content.parts:
            if part.text:
                response_text += part.text

    logger.info(f"[LLM RESPONSE] Length: {len(response_text)} chars")

    # FILTERING: Remove PII patterns
    filtered_text = response_text
    for pii_type, pattern in PII_PATTERNS.items():
        matches = re.findall(pattern, filtered_text)
        if matches:
            logger.warning(f"[FILTERED] Found {len(matches)} {pii_type} instan
            filtered_text = re.sub(pattern, f'[{pii_type.upper()}_REDACTED]',

```



```

# If we filtered anything, return modified response
if filtered_text != response_text:
    return types.GenerateContentResponse(
        candidates=[
            types.Candidate(
                content=types.Content(
                    parts=[types.Part(text=filtered_text)],
                    role="model"
                )
            )
        ]
    )

    return None # Use original response

def before_tool_callback(
    callback_context: CallbackContext,
    tool_name: str,
    args: Dict[str, Any]
) -> Optional[Dict[str, Any]]:
    """
    Called before executing a tool.

    Use Cases:
    1. Validation: Check arguments are valid
    2. Authorization: Check user permissions
    3. Rate limiting: Enforce usage limits
    4. Logging: Track tool usage

    Returns:
        None: Allow tool execution
        dict: Skip tool execution, use this result instead
    """
    logger.info(f"[TOOL CALL] {tool_name} with args: {args}")

    # VALIDATION: Check for negative values in generate_text
    if tool_name == 'generate_text':
        word_count = args.get('word_count', 0)
        if word_count <= 0 or word_count > 5000:
            logger.warning(f"[TOOL BLOCKED] Invalid word_count: {word_count}")
            return {
                'status': 'error',
                'message': f'Invalid word_count: {word_count}. Must be between'
            }

    # RATE LIMITING: Check tool usage quota

```

```

tool_count = callback_context.state.get(f'user:tool_{tool_name}_count', 0)
if tool_count >= 100: # Example limit
    logger.warning(f"[TOOL BLOCKED] Rate limit exceeded for {tool_name}")
    return {
        'status': 'error',
        'message': f'Rate limit exceeded for {tool_name}. Please try again'
    }

# Track tool usage
callback_context.state[f'user:tool_{tool_name}_count'] = tool_count + 1
callback_context.state['temp:last_tool'] = tool_name

return None # Allow tool execution

def after_tool_callback(
    callback_context: CallbackContext,
    tool_name: str,
    tool_response: Dict[str, Any]
) -> Optional[Dict[str, Any]]:
    """
    Called after tool execution completes.

    Use Cases:
    1. Logging: Record results
    2. Transformation: Standardize output format
    3. Caching: Store results for future use

    Returns:
        None: Use original tool result
        dict: Replace with modified result
    """
    logger.info(f"[TOOL RESULT] {tool_name}: {tool_response.get('status', 'unk')}")

    # Store last tool result for debugging
    callback_context.state['temp:last_tool_result'] = str(tool_response)

    # Could standardize all tool responses here
    # if 'status' not in tool_response:
    #     tool_response['status'] = 'success'

    return None # Use original result

# =====
# TOOLS
# =====

def generate_text(

```

```

    topic: str,
    word_count: int,
    tool_context: ToolContext
) -> Dict[str, Any]:
    """
    Generate text on a topic with specified word count.

    Args:
        topic: The subject to write about
        word_count: Desired number of words (1-5000)
    """
    # Tool would normally generate text here
    # For demo, just return metadata

    return {
        'status': 'success',
        'topic': topic,
        'word_count': word_count,
        'message': f'Generated {word_count}-word article on "{topic}"'
    }

def check_grammar(
    text: str,
    tool_context: ToolContext
) -> Dict[str, Any]:
    """
    Check grammar and provide corrections.

    Args:
        text: Text to check
    """
    # Simulate grammar checking
    issues_found = len(text.split()) // 10 # Fake: 1 issue per 10 words

    return {
        'status': 'success',
        'issues_found': issues_found,
        'message': f'Found {issues_found} potential grammar issues'
    }

def get_usage_stats(tool_context: ToolContext) -> Dict[str, Any]:
    """
    Get user's usage statistics from state.

    Shows how callbacks track metrics via state.
    """
    return {

```

```

        'status': 'success',
        'request_count': tool_context.state.get('user:request_count', 0),
        'llm_calls': tool_context.state.get('user:llm_calls', 0),
        'blocked_requests': tool_context.state.get('user:blocked_requests', 0)
        'tool_generate_text_count': tool_context.state.get('user:tool_generate
        'tool_check_grammar_count': tool_context.state.get('user:tool_check_gr
    }

# =====
# AGENT DEFINITION
# =====

root_agent = Agent(
    name="content_moderator",
    model="gemini-2.0-flash",

    description="""
Content moderation assistant with safety guardrails, validation, and monit
Demonstrates callback patterns for production-ready agents.
    """,

    instruction="""
You are a writing assistant that helps users create and refine content.

CAPABILITIES:
- Generate text on any topic with specified word count
- Check grammar and suggest corrections
- Provide usage statistics

SAFETY:
- You operate under strict content moderation policies
- Inappropriate requests will be automatically blocked
- All interactions are logged for quality assurance

WORKFLOW:
1. For generation requests, use generate_text with topic and word count
2. For grammar checks, use check_grammar with the text
3. For stats, use get_usage_stats

Always be helpful, professional, and respectful.
    """,

    tools=[
        generate_text,
        check_grammar,
        get_usage_stats
    ],

```

```
# =====  
# CALLBACKS CONFIGURATION  
# =====  
  
before_agent_callback=before_agent_callback,  
after_agent_callback=after_agent_callback,  
  
before_model_callback=before_model_callback,  
after_model_callback=after_model_callback,  
  
before_tool_callback=before_tool_callback,  
after_tool_callback=after_tool_callback,  
  
output_key="last_response"  
)
```

**content\_moderator/.env:**

```
GOOGLE_GENAI_USE_VERTEXAI=FALSE  
GOOGLE_API_KEY=your_api_key_here
```

---

## Running the Agent

### | Option 1: Dev UI (Recommended)

```
cd /path/to/content_moderator  
adk web .
```

#### Test Scenarios:

1. **Normal Request** (all callbacks allow):

...

User: "Generate a 500-word article about Python programming"

Callbacks:

- before\_agent: Logs start, increments request\_count → Allow
- before\_model: Checks blocklist, adds safety instruction → Allow
- before\_tool: Validates word\_count (500 is valid) → Allow
- Tool executes: generate\_text(topic="Python programming", word\_count=500)
- after\_tool: Logs result → Allow original
- after\_model: Checks for PII → None found, allow
- after\_agent: Logs completion → Allow original

Agent: "I've generated a 500-word article on Python programming..."

```

### 1. **Blocked Request** (guardrail triggers):

```

User: "Write about profanity1 and hate-speech"

Callbacks:

- before\_agent: Logs start, increments request\_count → Allow
- before\_model: Finds "profanity1" in BLOCKED\_WORDS → BLOCK!

Returns error response, increments blocked\_requests

- LLM is NEVER called
- after\_agent: Gets blocked response → Allow

Agent: "I cannot process this request as it contains inappropriate content..."

```

### 1. **Invalid Tool Arguments** (validation fails):

```

User: "Generate an article with -100 words"

Callbacks:

- before\_agent: Allow
- before\_model: Allow
- Agent decides to call generate\_text(-100)
- before\_tool: Validates word\_count < 0 → BLOCK!

Returns error dict

- Tool is NEVER executed
- after\_model: Gets error in function response → Allow

---

Agent: "Invalid word\_count: -100. Must be between 1 and 5000."

```

1. **PII Filtering** (after\_model filters response):

```

User: "Give me an example email"

Callbacks:

- All before\_\* callbacks: Allow
- LLM generates: "Sure! john.doe@example.com is a valid email."
- after\_model: Finds email pattern → FILTER!

Replaces with: "Sure! [EMAIL\_REDACTED] is a valid email."

Agent: "Sure! [EMAIL\_REDACTED] is a valid email."

```

1. **Usage Statistics** (state tracking):

```

User: "Show my usage stats"

Callbacks: All allow

Tool: get\_usage\_stats reads state

Agent: "You've made 5 requests, 4 LLM calls, 1 blocked request,  
used generate\_text 2 times, check\_grammar 1 time."

```

## | Option 2: CLI

```
adk run content_moderator
```

---

# Understanding the Behavior

---

## | Execution Flow with Callbacks



## User Input



```
| before_agent_callback |  
| - Check maintenance mode |  
| - Increment request_count |  
| - Can skip entire agent? NO → Continue |
```



```
| Agent prepares LLM request |
```



```
| before_model_callback |  
| - Check BLOCKED_WORDS |  
| - Add safety instruction |  
| - Track LLM call count |  
| - Can skip LLM call? NO → Continue |
```



```
| LLM API Call (Gemini 2.0 Flash) |
```



```
| after_model_callback |  
| - Scan for PII patterns |  
| - Filter email/phone/SSN |  
| - Can replace response? NO → Continue |
```

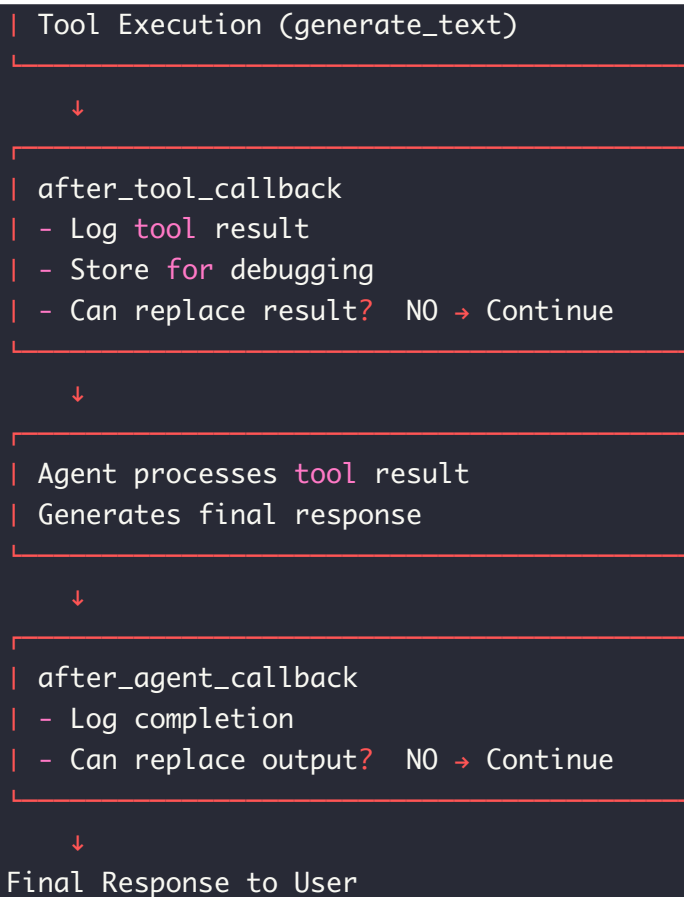


```
| Agent decides to call tool |  
| (e.g., generate_text) |
```



```
| before_tool_callback |  
| - Validate arguments (word_count) |  
| - Check rate limits |  
| - Track tool usage |  
| - Can skip tool? NO → Continue |
```





## Events Tab View

In `adk web`, the Events tab shows:

### Normal Flow:

```

Event 1: user_request
Event 2: before_agent_callback executed
Event 3: state_update (user:request_count = 1)
Event 4: before_model_callback executed
Event 5: state_update (user:llm_calls = 1)
Event 6: llm_request sent
Event 7: llm_response received
Event 8: after_model_callback executed
Event 9: before_tool_callback executed
Event 10: state_update (user:tool_generate_text_count = 1)
Event 11: tool_call (generate_text)
Event 12: tool_result
Event 13: after_tool_callback executed
Event 14: after_agent_callback executed
Event 15: final_response
  
```

**Blocked Flow** (guardrail triggered):

```
Event 1: user_request
Event 2: before_agent_callback executed
Event 3: state_update (user:request_count = 1)
Event 4: before_model_callback executed
Event 5: BLOCKED! (found blocked word)
Event 6: state_update (user:blocked_requests = 1)
Event 7: synthetic_llm_response (from callback)
Event 8: after_agent_callback executed
Event 9: final_response (error message)
```

**Note:** No `llm_request` or `tool_call` events when blocked!

## How It Works: Callback Patterns Deep Dive

### | Pattern 1: Guardrails (Block Before Execution)

```
def before_model_callback(callback_context, llm_request):
    # Check condition
    if contains_blocked_content(llm_request):
        # Return response object to SKIP LLM call
        return types.GenerateContentResponse(...)

    return None # Allow LLM call
```

**Why it works:**

- Returning an object tells ADK: "I've got the response, don't call the LLM"
- Saves API costs and latency
- LLM never sees inappropriate content

## Pattern 2: Validation (Check Arguments)

```
def before_tool_callback(callback_context, tool_name, args):  
    # Validate arguments  
    if args['word_count'] <= 0:  
        # Return error dict to SKIP tool execution  
        return {'status': 'error', 'message': '...'}  
  
    return None # Allow tool execution
```

### Why it works:

- Prevents tool from running with invalid arguments
- Tool function never executes
- Returns error as if tool ran

## Pattern 3: Logging (Observe All Operations)

```
def before_tool_callback(callback_context, tool_name, args):  
    logger.info(f"Tool: {tool_name}, Args: {args}")  
    return None # Just observe, don't block  
  
def after_tool_callback(callback_context, tool_name, result):  
    logger.info(f"Result: {result}")  
    return None # Just observe, don't modify
```

### Why it works:

- Callbacks see all operations
- Returning `None` means "proceed normally"
- Creates audit trail without changing behavior

## Pattern 4: Modification (Transform Data)

```
def before_model_callback(callback_context, llm_request):  
    # Modify request in place  
    llm_request.config.system_instruction += "\nBe concise."  
    return None # Allow modified request to proceed  
  
def after_model_callback(callback_context, llm_response):  
    # Replace response  
    filtered_text = remove_pii(llm_response.text)  
    return types.GenerateContentResponse(...)
```

### Why it works:

- `before_*`: Modify request, return `None` to use modified version
- `after_*`: Return new object to replace original

## Pattern 5: State Tracking (Metrics & Analytics)

```
def before_model_callback(callback_context, llm_request):  
    # Track metrics in state  
    count = callback_context.state.get('user:llm_calls', 0)  
    callback_context.state['user:llm_calls'] = count + 1  
    return None
```

### Why it works:

- Callbacks have access to `callback_context.state`
- Changes are automatically persisted in events
- Can track anything: counts, timestamps, usage patterns

## Key Takeaways

1. **Callbacks = Execution Hooks:**
2. Run at specific points in agent lifecycle
3. Provide context (session, state, request/response)

---

4. Can observe, modify, or block operations

5. **Return** `None` **vs Object**:

6. `None` → Proceed normally (allow default)

7. Object → Override/skip operation

8. **Callback Types Serve Different Purposes:**

9. `before_agent` → Authorization, maintenance checks

10. `before_model` → Guardrails, request modification

11. `after_model` → Response filtering, PII removal

12. `before_tool` → Argument validation, rate limiting

13. `after_tool` → Result logging, caching

14. `after_agent` → Final output validation

15. **State Management in Callbacks:**

16. Use `callback_context.state` for metrics

17. Changes are automatically tracked

18. Can use `user:`, `app:`, `temp:` prefixes

19. **Callbacks Run Synchronously:**

20. Keep them fast (no long-running operations)

21. Heavy processing should be offloaded

22. Each callback adds latency

---

## Best Practices

---

### | Design Principles

**DO:**

- ✓ Keep callbacks focused (single purpose)
- ✓ Use descriptive names: `check_profanity_guard` not `callback1`
- ✓ Log important decisions

- ✓ Handle errors gracefully (try/except)
- ✓ Document what each callback does

**DON'T:**

- ✗ Do heavy computation in callbacks
- ✗ Make external API calls (if possible)
- ✗ Create monolithic callbacks (do everything in one)
- ✗ Forget to return `None` when allowing default behavior

## Error Handling

```
def before_model_callback(callback_context, llm_request):
    try:
        # Check for blocked content
        if check_blocklist(llm_request):
            return create_blocked_response()
        return None
    except Exception as e:
        logger.error(f"Callback error: {e}")
        # Decide: Block request or allow?
        return None # Allow on error (fail open)
        # OR
        # return create_error_response() # Block on error (fail closed)
```

## State Management

```
def track_usage(callback_context):
    # Use descriptive keys
    key = f'user:{callback_context.user_id}:llm_calls'

    # Initialize if not exists
    count = callback_context.state.get(key, 0)

    # Update
    callback_context.state[key] = count + 1
```

## Testing Callbacks

```
# Unit test with mock context
def test_before_model_blocks_profanity():
    mock_context = MockCallbackContext()
    mock_request = create_request_with_profanity()

    result = before_model_callback(mock_context, mock_request)

    assert result is not None # Should block
    assert "inappropriate content" in result.text
```

## Common Issues & Troubleshooting

### Issue 1: Callback Not Running

**Problem:** Set `before_model_callback` but it never executes

**Solutions:**

1. Check callback is set on Agent:

```
python root_agent = Agent( ..., before_model_callback=my_callback # Must be set! )
```

2. Verify callback signature matches:

```
python # Correct signature def before_model_callback( callback_context:
CallbackContext, llm_request: types.GenerateContentRequest ) ->
Optional[types.GenerateContentResponse]:
```

3. Check agent type (model callbacks only work on LlmAgent)

### Issue 2: Callback Blocks Everything

**Problem:** All requests get blocked unexpectedly

**Solutions:**

1. Check return value:



```
```python
# BAD: Always returns object (blocks all)
def before_model_callback(ctx, req):
    return types.GenerateContentResponse(...)

# GOOD: Returns None to allow
def before_model_callback(ctx, req):
    if should_block(req):
        return types.GenerateContentResponse(...)
    return None # Allow!
```
```

1. Add debug logging to understand flow
2. Test callback logic in isolation

## | Issue 3: State Changes Not Persisting

**Problem:** Set state in callback but it's gone later

**Solutions:**

1. Use correct context:

```
```python
# BAD: Wrong context type
def my_callback(context, ...):
    context.state['key'] = 'value' # Wrong!

# GOOD: Use callback_context
def my_callback(callback_context: CallbackContext, ...):
    callback_context.state['key'] = 'value' # Right!
```
```

1. Ensure persistent SessionService for cross-session state
2. Remember `temp:` prefix is never persisted

## | Issue 4: Callback Causes Errors

**Problem:** Agent crashes when callback runs

---

**Solutions:**

1. Add error handling:

```
python def before_model_callback(ctx, req): try: # Callback logic ... except  
Exception as e: logger.error(f"Callback failed: {e}") return None # Allow on  
error
```

2. Check all required imports are present
  3. Validate context/request objects before accessing
  4. Test callbacks independently
- 

## Real-World Applications

---

### 1. Content Moderation Platform

- `before_model_callback`: Block hate speech, profanity, illegal content
- `after_model_callback`: Filter PII, redact sensitive info
- `before_tool_callback`: Validate image URLs, check file sizes
- State tracking: Usage metrics, violation counts

### 2. Enterprise Support Agent

- `before_agent_callback`: Check user permissions, enforce business hours
- `before_tool_callback`: Validate customer IDs, check authorization
- `after_tool_callback`: Log all database queries for audit
- State tracking: Support ticket counts, resolution times

### 3. Healthcare Assistant

- `before_model_callback`: Enforce HIPAA compliance, block PHI in logs
- `after_model_callback`: Add medical disclaimers to all responses
- `before_tool_callback`: Verify patient consent before data access
- State tracking: Consultation counts, compliance checks


---

## 4. Financial Advisor Bot

- `before_agent_callback`: Check market hours, trading permissions
  - `before_tool_callback`: Validate transaction amounts, check fraud patterns
  - `after_tool_callback`: Encrypt sensitive financial data
  - State tracking: Transaction counts, risk scores
- 

## Next Steps

---

 **Tutorial 10: Evaluation & Testing** - Learn systematic testing of agent behavior and callbacks

### Exercises:

1. Add `before_agent_callback` to check user rate limits
  2. Implement caching in `before_model_callback` using state
  3. Create `after_tool_callback` that saves all results to a database
  4. Build custom PII filter that handles additional patterns
- 






## Working Implementation

---

A complete, production-ready implementation of this tutorial is available at:

 [tutorial\\_implementation/tutorial09/content\\_moderator/](https://github.com/raphaelmansuy/adk_training/tree/main/tutorial_implementation/tutorial09/content_moderator/) ([https://github.com/raphaelmansuy/adk\\_training/tree/main/tutorial\\_implementation/tutorial09/content\\_moderator/](https://github.com/raphaelmansuy/adk_training/tree/main/tutorial_implementation/tutorial09/content_moderator/))

## What's Included

-  **Complete Agent:** Content moderator with all 6 callback types
  -  **Comprehensive Tests:** 11 test cases covering all callback scenarios
  -  **Production Features:** Logging, metrics, PII filtering, rate limiting
  -  **Developer Tools:** Makefile, requirements.txt, detailed README
  -  **Security:** Blocklist filtering, input validation, guardrails
-

## Quick Start

```
cd tutorial_implementation/tutorial09/content_moderator
make setup
cp .env.example .env # Add your GOOGLE_API_KEY
make test           # Run comprehensive tests
make dev            # Start ADK web interface
```

## Features Demonstrated

### Guardrails & Safety:

- Blocks profanity before LLM calls
- Filters PII from responses
- Validates tool arguments
- Rate limiting protection

### Monitoring & Observability:

- Complete audit logging
- Usage metrics tracking
- State management across sessions
- Performance monitoring

### Callback Patterns:

- All 6 callback types implemented
- Control flow examples (block vs allow)
- State manipulation patterns
- Error handling best practices

## Test Coverage

The implementation includes comprehensive tests for:

- Maintenance mode blocking
- Request counting
- Profanity filtering

- Safety instruction injection
- PII redaction
- Tool validation
- Rate limiting
- Result logging
- Tool functionality
- Usage statistics

Run `make test` to see all callback patterns in action!

---

## Further Reading

---

- [Callbacks Documentation](https://google.github.io/adk-docs/callbacks/) (https://google.github.io/adk-docs/callbacks/)
  - [Types of Callbacks](https://google.github.io/adk-docs/callbacks/types-of-callbacks/) (https://google.github.io/adk-docs/callbacks/types-of-callbacks/)
  - [Callback Design Patterns](https://google.github.io/adk-docs/callbacks/design-patterns-and-best-practices/) (https://google.github.io/adk-docs/callbacks/design-patterns-and-best-practices/)
  - [Safety & Security Guide](https://google.github.io/adk-docs/safety/) (https://google.github.io/adk-docs/safety/)
  - [Context Objects Reference](https://google.github.io/adk-docs/context/) (https://google.github.io/adk-docs/context/)
- 

**Congratulations!** You now understand how to use callbacks for guardrails, monitoring, and control flow in production agents. This enables safe, compliant, and observable AI systems.

---

Generated on 2025-10-19 17:56:27 from 09\_callbacks\_guardrails.md

Source: Google ADK Training Hub