Google ADK: Multiple Human-in-the-Loop Implementation Guide

📋 Overview

This guide shows how to implement **multiple sequential human-in-the-loop approvals** in Google ADK workflows. The key insight is that ADK requires careful **global state management** to handle multiple long-running functions in sequence.

🏗️ Architecture

Core Components

1. **Global State Variables** - Track workflow state across function calls
2. **Long-Running Approval Functions** - Display forms and pause execution
3. **Unified Resume Function** - Handles any approval type and cleans state
4. **Robust Workflow Handler** - Detects multiple approvals with fallback logic

Data Flow

Initial Workflow → First Approval (pause) → Resume → Second Approval (pause) → Resume → Complete

🔧 Implementation

1. Global State Setup

```
#
================================================================================
# GLOBAL STATE FOR WORKFLOW MANAGEMENT
#
================================================================================
PENDING_APPROVALS = {}          # Unified approval storage
CURRENT_RUNNER = None           # ADK runner instance
CURRENT_SESSION = None          # ADK session instance
CURRENT_LONG_RUNNING_CALL = None # Current paused function call
```

CURRENT_FUNCTION_RESPONSE = None # Current pending response

2. Long-Running Approval Functions

Pattern for Any Approval Type

```
def approval_function(purpose: str, tool_context: ToolContext) -> dict:
    """Template for any approval function"""

    # 1. Get data from context
    data = tool_context.state.get("your_data_key")
    if not data:
        return {"status": "error", "message": "No data found"}

    # 2. Generate unique approval ID
    approval_id = f"your_approval_{int(time.time())}"

    # 3. Store in unified PENDING_APPROVALS
    PENDING_APPROVALS[approval_id] = {
        "status": "pending",
        "your_data": data,  # Store the specific data needed
        "purpose": purpose,
        "created_at": time.time()
    }

    # 4. Store in context using CONSISTENT keys
    tool_context.state["approval_id"] = approval_id        # Same key for all
    tool_context.state["approval_status"] = "pending"     # Same key for all

    # 5. Display your approval UI
    _display_your_approval_form(data, approval_id)

    # 6. Return pending status to pause ADK
    return {
        "status": "pending",
        "approval_id": approval_id,
        "purpose": purpose,
        "message": "Approval form displayed. Agent execution paused."
    }
```

Minimal Campaign Approval Example

```
def campaign_approval(purpose: str, tool_context: ToolContext) -> dict:
    campaign_data = tool_context.state.get("campaign_data")
```

```
    approval_id = f"campaign_{int(time.time())}"

    PENDING_APPROVALS[approval_id] = {
        "status": "pending",
        "campaign_data": campaign_data,
        "purpose": purpose,
        "created_at": time.time()
    }

    tool_context.state["approval_id"] = approval_id
    tool_context.state["approval_status"] = "pending"

    # Your UI code here
    print(f"Campaign approval required: {campaign_data}")
    display_campaign_form(campaign_data, approval_id)

    return {
        "status": "pending",
        "approval_id": approval_id,
        "purpose": purpose,
        "message": "Campaign approval form displayed."
    }
```

Minimal Content Approval Example

```
def content_approval(purpose: str, tool_context: ToolContext) -> dict:
    video_path = tool_context.state.get("final_video")
    approval_id = f"content_{int(time.time())}"

    PENDING_APPROVALS[approval_id] = {
        "status": "pending",
        "video_path": video_path,
        "purpose": purpose,
        "created_at": time.time()
    }

    tool_context.state["approval_id"] = approval_id
    tool_context.state["approval_status"] = "pending"

    # Your UI code here
    print(f"Content approval required: {video_path}")
    display_video_form(video_path, approval_id)

    return {
```

```
        "status": "pending",
        "approval_id": approval_id,
        "purpose": purpose,
        "message": "Content approval form displayed."
    }
```

3. Unified Resume Function

```
async def resume_workflow():
    """Handles ANY approval type and properly cleans state"""
    global CURRENT_RUNNER, CURRENT_SESSION, CURRENT_LONG_RUNNING_CALL,
CURRENT_FUNCTION_RESPONSE

    if not CURRENT_FUNCTION_RESPONSE:
        print("❌ No paused workflow to resume.")
        return

    approval_id = CURRENT_FUNCTION_RESPONSE.response.get('approval_id')

    if approval_id not in PENDING_APPROVALS:
        print(f"❌ Approval ID {approval_id} not found")
        return

    approval_info = PENDING_APPROVALS[approval_id]
    if approval_info["status"] != "approved":
        print("⏳ Please approve/reject first.")
        return

    print("✅ Approval confirmed! Resuming workflow...")

    # Handle different approval types by checking what data exists
    from google.adk.events import Event, EventActions
    state_changes = {}

    if "campaign_data" in approval_info:
        state_changes["campaign_data"] = approval_info["campaign_data"]
    if "video_path" in approval_info:
        state_changes["approved_video"] = approval_info["video_path"]
    # Add more data types as needed

    # Update session state
    actions = EventActions(state_delta=state_changes)
    event = Event(invocation_id="approval_update", author="system",
                  actions=actions, timestamp=time.time())
```

```
    await CURRENT_RUNNER.session_service.append_event(CURRENT_SESSION, event)

    # 🚨 CRITICAL: Clean state BEFORE resuming
    CURRENT_LONG_RUNNING_CALL = None
    CURRENT_FUNCTION_RESPONSE = None

    # Resume workflow
    follow_up = types.Content(role='user', parts=[types.Part(text="Approved. Continue.")])

    # Process resumed events (important for detecting next approvals)
    resume_events = CURRENT_RUNNER.run_async(
        user_id="user", session_id=CURRENT_SESSION.id, new_message=follow_up)

    async for event in resume_events:
        # Same detection logic as main workflow
        current_call = get_long_running_function_call(event)
        if current_call:
            CURRENT_LONG_RUNNING_CALL = current_call

        if CURRENT_LONG_RUNNING_CALL:
            response = get_function_response(event, CURRENT_LONG_RUNNING_CALL.id)
            if response and response.response.get('status') == 'pending':
                CURRENT_FUNCTION_RESPONSE = response
                print("🛑 Next approval detected! Call resume_workflow() again.")
                return

    print("🎉 Workflow completed!")
```

4. Robust Workflow Handler

```
async def run_multi_approval_workflow(runner, session, user_message, events_list):
    """Handles multiple sequential approvals with proper state management"""
    global CURRENT_RUNNER, CURRENT_SESSION, CURRENT_LONG_RUNNING_CALL,
CURRENT_FUNCTION_RESPONSE

    CURRENT_RUNNER = runner
    CURRENT_SESSION = session

    events_async = runner.run_async(user_id="user", session_id=session.id,
new_message=user_message)

    async for event in events_async:
        events_list.append(event)
```

```
    # Check for long-running function calls
    current_call = get_long_running_function_call(event)
    if current_call:
        CURRENT_LONG_RUNNING_CALL = current_call

    # Check for function responses
    if CURRENT_LONG_RUNNING_CALL:
        response = get_function_response(event, CURRENT_LONG_RUNNING_CALL.id)
        if response:
            status = response.response.get('status')
            CURRENT_FUNCTION_RESPONSE = response

            if status == "pending":
                print("🛑 Approval required! Call: await resume_workflow()")
                return  # PAUSE

            elif status in ["approved", "rejected"]:
                # 🚨 CRITICAL: Reset state for next approval
                CURRENT_LONG_RUNNING_CALL = None
                CURRENT_FUNCTION_RESPONSE = None

    # Fallback: Check for approval agents completing with pending approvals
    if (event.is_final_response() and
        event.author.endswith('_approver')):  # Any agent ending in _approver

        recent_pending = [aid for aid, info in PENDING_APPROVALS.items()
                          if info['status'] == 'pending' and
                          (time.time() - info['created_at']) < 10]

        if recent_pending:
            latest = max(recent_pending, key=lambda x: PENDING_APPROVALS[x]['created_at'])
            CURRENT_FUNCTION_RESPONSE = type('MockResponse', (), {
                'response': {'approval_id': latest, 'status': 'pending'}
            })()
            print("🛑 Fallback approval detected! Call: await resume_workflow()")
            return  # PAUSE

print("Workflow completed")
```

🎯 Usage Examples

Basic Two-Step Approval Workflow

```
# 1. Set up agents
campaign_approver = Agent(
    name="campaign_approver",
    tools=[LongRunningFunctionTool(func=campaign_approval)]
)

content_approver = Agent(
    name="content_approver",
    tools=[LongRunningFunctionTool(func=content_approval)]
)

workflow = SequentialAgent(
    name="approval_workflow",
    sub_agents=[campaign_approver, content_approver]
)

# 2. Run workflow
await run_multi_approval_workflow(runner, session, user_message, events)

# 3. Handle first approval
# User sees campaign form, clicks approve
await resume_workflow()

# 4. Handle second approval
# User sees content form, clicks approve
await resume_workflow()

# 5. Workflow completes
```

Three-Step Approval Workflow

```
# Add a third approval type
def budget_approval(purpose: str, tool_context: ToolContext) -> dict:
    budget_data = tool_context.state.get("budget_breakdown")
    approval_id = f"budget_{int(time.time())}"

    PENDING_APPROVALS[approval_id] = {
        "status": "pending",
        "budget_data": budget_data,
        "purpose": purpose,
        "created_at": time.time()
    }
```

```
    tool_context.state["approval_id"] = approval_id
    tool_context.state["approval_status"] = "pending"

    display_budget_form(budget_data, approval_id)

    return {
        "status": "pending",
        "approval_id": approval_id,
        "purpose": purpose,
        "message": "Budget approval required."
    }

# Workflow with three approvals
workflow = SequentialAgent(
    name="three_step_workflow",
    sub_agents=[
        campaign_approver,   # First approval
        budget_approver,     # Second approval
        content_approver     # Third approval
    ]
)

# Usage: Same pattern, just call resume_workflow() three times
```

⚠️ Critical Success Factors

1. Consistent Context Keys

```
# ✅ DO: Use same keys for all approval types
tool_context.state["approval_id"] = approval_id
tool_context.state["approval_status"] = "pending"

# ❌ DON'T: Use different keys per approval type
tool_context.state["campaign_approval_id"] = approval_id    # Different key
tool_context.state["content_approval_id"] = approval_id     # Different key
```

2. Unified Data Storage

```
# ✅ DO: Use single approval dictionary
PENDING_APPROVALS[approval_id] = {
    "status": "pending",
    "campaign_data": data,  # Type-specific data
```

```
    # OR
    "video_path": path,     # Type-specific data
}

# ❌ DON'T: Use separate dictionaries
PENDING_CAMPAIGN_APPROVALS = {}  # Separate dict
PENDING_CONTENT_APPROVALS = {}   # Separate dict
```

3. Proper State Cleanup

```
# ✅ DO: Reset state after each approval
if status in ["approved", "rejected"]:
    CURRENT_LONG_RUNNING_CALL = None
    CURRENT_FUNCTION_RESPONSE = None

# ❌ DON'T: Leave state dirty between approvals
# (No cleanup - next approval can't be detected)
```

4. Resume Function State Management

```
# ✅ DO: Clean state BEFORE resuming
CURRENT_LONG_RUNNING_CALL = None
CURRENT_FUNCTION_RESPONSE = None
# Then resume...

# ❌ DON'T: Clean state AFTER resuming
# Resume first, then clean (too late!)
```

🐛 Common Issues & Solutions

Issue: Second approval doesn't pause

Cause: State not cleaned after first approval
Solution: Reset `CURRENT_LONG_RUNNING_CALL` and `CURRENT_FUNCTION_RESPONSE` after processing

Issue: Resume function only works for first approval type

Cause: Hardcoded to specific data type
Solution: Use unified detection based on what data exists in approval_info

Issue: Approval forms display but workflow doesn't pause

Cause: Long-running function detection failing
Solution: Add fallback detection for approval agent completion

Issue: ValueError about function call ID not found

Cause: Trying to resume with stale function response
Solution: Ensure `CURRENT_FUNCTION_RESPONSE` matches active approval

🎉 Best Practices

1. **Use descriptive approval IDs**: `campaign_approval_1234567890`
2. **Add timestamps**: Track when approvals were created
3. **Implement timeouts**: Clean up old pending approvals
4. **Add debug functions**: Check state between approvals
5. **Test sequentially**: Verify each approval works before adding more
6. **Use fallback detection**: Don't rely only on long-running function detection
7. **Keep UI simple**: Focus on core approve/reject functionality first

📊 Debug Functions

```
def debug_approval_state():
    """Check current workflow state"""
    print(f"Pending approvals: {len(PENDING_APPROVALS)}")
    print(f"Current call: {CURRENT_LONG_RUNNING_CALL.name if
CURRENT_LONG_RUNNING_CALL else 'None'}")
    print(f"Current response: {'Set' if CURRENT_FUNCTION_RESPONSE else 'None'}")

    for aid, info in PENDING_APPROVALS.items():
        data_type = next((k for k in info.keys() if k.endswith('_data') or k == 'video_path'),
'unknown')
        print(f"  {aid}: {info['status']} ({data_type})")

def cleanup_old_approvals(max_age_minutes=60):
    """Clean up old pending approvals"""
    current_time = time.time()
    to_remove = []

    for aid, info in PENDING_APPROVALS.items():
        age_minutes = (current_time - info['created_at']) / 60
        if age_minutes > max_age_minutes:
            to_remove.append(aid)
```

```
    for aid in to_remove:
        del PENDING_APPROVALS[aid]
        print(f"Cleaned up old approval: {aid}")
```

This guide provides a robust foundation for implementing multiple human-in-the-loop approvals in Google ADK workflows. The key is careful state management and unified data handling patterns.