Tutorial 08: State and Memory - Persistent Agent Context

Difficulty: intermediate
Reading Time: 1 hour
Tags: intermediate, state, memory, persistence, context
Description: Manage agent state and memory across conversations using session, user, and application-level persistence for stateful interactions.

Tutorial 08: State Memory - Managing Conversation Context and Data

💡 **View the complete working implementation and test suite here. (https://github.com/raphaelmansuy/adk_training/tree/main/tutorial_implementation/tutorial08/README.md)**

Overview

Learn how to build agents that remember information across interactions using **session state** and **long-term memory**. This tutorial demonstrates a personal tutor system that tracks user progress, preferences, and learning history.

What You'll Build: A personalized learning assistant that:

- Remembers user preferences (language, difficulty level)
- Tracks progress across sessions (topics covered, quiz scores)

- Uses temporary state for calculations
- Searches past learning sessions for context
- Adapts teaching based on history

Why It Matters: Most production agents need to maintain context beyond a single conversation. State management and memory enable personalized, context-aware experiences.

Prerequisites

- Python 3.9+
- `google-adk` installed (`pip install google-adk`)
- Google API key (see Authentication Setup (https://google.github.io/adk-docs/get-started/quickstart/))
- Completed Tutorials 01-02 (basics of agents and tools)
- Understanding of session concepts

Core Concepts

Session State (`session.state`)

The agent's **scratchpad** - a key-value dictionary for conversation-level data.

State Scoping with Prefixes:

Prefix	Scope	Persistence	Example Use Case
None	Current session	SessionService dependent	`state['current_topic'] = 'python'` - Task progress
`user:`	All sessions for user	Persistent	`state['user:preferred_language'] = 'en'` - User preferences
`app:`	All users/ sessions	Persistent	`state['app:course_catalog'] = [...]` - Global settings
`temp:`	Current invocation only	**Never persisted**	`state['temp:quiz_score'] = 85` - Temporary calculations

Key Points:

- `temp:` state is **discarded** after invocation completes
- `temp:` state is **shared** across all sub-agents in same invocation
- `user:` and `app:` require persistent SessionService (Database/VertexAI)
- Use `output_key` or `context.state` to update safely

Memory Service

Long-term knowledge beyond current session - like a **searchable archive**.

Two Implementations:

1. **InMemoryMemoryService**: Keyword search, no persistence (dev/test)
2. **VertexAiMemoryBankService**: Semantic search, LLM-powered, persistent (production)

Workflow:

1. User interacts with agent (session)
2. Call `add_session_to_memory(session)` to save
3. Later, agent searches: `search_memory(query)`
4. Memory returns relevant past interactions
5. Agent uses retrieved context

Use Case: Personal Learning Tutor

Scenario: Build a tutor that:

- Stores user preferences (language, difficulty)
- Tracks what topics you've studied
- Remembers your quiz performance
- Searches past lessons when you ask questions
- Adapts explanations based on your level

State Strategy:

- `user:language` → Preference (persistent across sessions)
- `user:difficulty_level` → Preference (beginner/intermediate/advanced)
- `user:topics_covered` → List of completed topics
- `user:quiz_scores` → History of quiz performance
- `current_topic` → What we're studying now (session-level)
- `temp:quiz_answers` → Answers during quiz (discarded after)

Implementation

Project Structure

```
personal_tutor/
├── __init__.py          # Imports agent
├── agent.py             # Agent definition
└── .env.example         # API key template
```

Complete Code

personal_tutor/init**.py**:

```
from .agent import root_agent

__all__ = ['root_agent']
```

personal_tutor/agent.py:

```
"""
Personal Learning Tutor - Demonstrates State & Memory Management

This agent uses:
- user: prefix for persistent preferences (language, difficulty)
- Session state for current topic tracking
- temp: prefix for temporary quiz calculations
- Memory service for retrieving past learning sessions
"""

from google.adk.agents import Agent
from google.adk.tools.tool_context import ToolContext
from typing import Dict, Any

# ============================================================================
# TOOLS: State Management & Memory Operations
# ============================================================================

def set_user_preferences(
    language: str,
    difficulty_level: str,
    tool_context: ToolContext
) -> Dict[str, Any]:
    """
    Set user learning preferences (stored persistently).

    Args:
        language: Preferred language (en, es, fr, etc.)
        difficulty_level: beginner, intermediate, or advanced
    """
    # Use user: prefix for persistent cross-session storage
    tool_context.state['user:language'] = language
    tool_context.state['user:difficulty_level'] = difficulty_level

    return {
        'status': 'success',
        'message': f'Preferences saved: {language}, {difficulty_level} level'
    }

def record_topic_completion(
    topic: str,
    quiz_score: int,
    tool_context: ToolContext
) -> Dict[str, Any]:
    """
    Record that user completed a topic (stored persistently).
```

```
    Args:
        topic: Topic name (e.g., "Python Basics", "Data Structures")
        quiz_score: Score out of 100
    """
    # Get existing lists or create new ones
    topics = tool_context.state.get('user:topics_covered', [])
    scores = tool_context.state.get('user:quiz_scores', {})

    # Update persistent user state
    if topic not in topics:
        topics.append(topic)
    scores[topic] = quiz_score

    tool_context.state['user:topics_covered'] = topics
    tool_context.state['user:quiz_scores'] = scores

    return {
        'status': 'success',
        'topics_count': len(topics),
        'message': f'Recorded: {topic} with score {quiz_score}/100'
    }

def get_user_progress(tool_context: ToolContext) -> Dict[str, Any]:
    """
    Get user's learning progress summary.

    Returns persistent user data across all sessions.
    """
    # Read persistent user state
    language = tool_context.state.get('user:language', 'en')
    difficulty = tool_context.state.get('user:difficulty_level', 'beginner')
    topics = tool_context.state.get('user:topics_covered', [])
    scores = tool_context.state.get('user:quiz_scores', {})

    # Calculate average score
    avg_score = sum(scores.values()) / len(scores) if scores else 0

    return {
        'status': 'success',
        'language': language,
        'difficulty_level': difficulty,
        'topics_completed': len(topics),
        'topics': topics,
        'average_quiz_score': round(avg_score, 1),
        'all_scores': scores
    }
```

```python
def start_learning_session(
    topic: str,
    tool_context: ToolContext
) -> Dict[str, Any]:
    """
    Start a new learning session for a topic.

    Uses session state (no prefix) to track current topic.
    """
    # Session-level state (persists within this session only)
    tool_context.state['current_topic'] = topic
    tool_context.state['session_start_time'] = 'now'  # Simplified

    # Get user's difficulty level for personalization
    difficulty = tool_context.state.get('user:difficulty_level', 'beginner')

    return {
        'status': 'success',
        'topic': topic,
        'difficulty_level': difficulty,
        'message': f'Started learning session: {topic} at {difficulty} level'
    }

def calculate_quiz_grade(
    correct_answers: int,
    total_questions: int,
    tool_context: ToolContext
) -> Dict[str, Any]:
    """
    Calculate quiz grade using temporary state.

    Demonstrates temp: prefix for invocation-scoped data.
    """
    # Store intermediate calculation in temp state (discarded after invocation
    percentage = (correct_answers / total_questions) * 100
    tool_context.state['temp:raw_score'] = correct_answers
    tool_context.state['temp:quiz_percentage'] = percentage

    # Determine grade letter
    if percentage >= 90:
        grade = 'A'
    elif percentage >= 80:
        grade = 'B'
    elif percentage >= 70:
        grade = 'C'
    elif percentage >= 60:
```

```
        grade = 'D'
    else:
        grade = 'F'

    return {
        'status': 'success',
        'score': f'{correct_answers}/{total_questions}',
        'percentage': round(percentage, 1),
        'grade': grade,
        'message': f'Quiz grade: {grade} ({percentage:.1f}%)'
    }

def search_past_lessons(
    query: str,
    tool_context: ToolContext
) -> Dict[str, Any]:
    """
    Search memory for relevant past learning sessions.

    This demonstrates memory service integration.
    In production, this would use MemoryService.search_memory().
    """
    # NOTE: This is a simplified simulation
    # Real implementation would use:
    # memory_service = tool_context.memory_service
    # results = await memory_service.search_memory(
    #     app_name=tool_context.app_name,
    #     user_id=tool_context.user_id,
    #     query=query
    # )

    # Simulated memory search results
    topics = tool_context.state.get('user:topics_covered', [])
    relevant = [t for t in topics if query.lower() in t.lower()]

    if relevant:
        return {
            'status': 'success',
            'found': True,
            'relevant_topics': relevant,
            'message': f'Found {len(relevant)} past sessions related to "{quer
        }
    else:
        return {
            'status': 'success',
            'found': False,
            'message': f'No past sessions found for "{query}"'
```

```
        }

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

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

    description="""
    Personal learning tutor that tracks your progress, preferences, and learni
    Uses state management and memory to provide personalized education.
    """,

    instruction="""
    You are a personalized learning tutor with memory of the user's progress.

    CAPABILITIES:
    - Set and remember user preferences (language, difficulty level)
    - Track completed topics and quiz scores across sessions
    - Start new learning sessions on specific topics
    - Calculate quiz grades and store results
    - Search past learning sessions for context
    - Adapt teaching based on user's level and history

    STATE MANAGEMENT:
    - User preferences stored with user: prefix (persistent)
    - Current session tracked with session state
    - Temporary calculations use temp: prefix (discarded after)

    TEACHING APPROACH:
    1. Check user's difficulty level and adapt explanations
    2. Reference past topics when relevant
    3. Track progress and celebrate achievements
    4. Provide personalized recommendations based on history

    WORKFLOW:
    1. If new user, ask about preferences (language, difficulty)
    2. For learning requests:
       - Start a session with start_learning_session
       - Teach the topic at appropriate level
       - End with a quiz
    3. Record completion with quiz score
    4. Search past lessons when user asks about previous topics

    Always be encouraging and adapt to the user's learning pace!
```

```
    """,

    tools=[
        set_user_preferences,
        record_topic_completion,
        get_user_progress,
        start_learning_session,
        calculate_quiz_grade,
        search_past_lessons
    ],

    # Save final response to session state
    output_key="last_tutor_response"
)
```

personal_tutor/.env:

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

Running the Agent

Option 1: Dev UI (Recommended)

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

Workflow to Test:

1. **Set Preferences** (creates `user:` state):

```

User: "Set my language to English and difficulty to intermediate"
```

Agent: [calls set_user_preferences]
"Great! I've saved your preferences: English, intermediate level."
```

1. **Start Learning** (creates session state):

```
User: "Teach me about Python functions"

Agent: [calls start_learning_session('Python functions')]
[Explains Python functions at intermediate level]
```

1. **Take Quiz** (uses `temp:` state):

```
User: "I got 8 out of 10 questions correct"

Agent: [calls calculate_quiz_grade(8, 10)]
"Excellent! You scored 80% (B grade) on the quiz."
```

1. **Record Completion** (updates `user:` state):

`Agent: [calls record_topic_completion('Python functions', 80)] "I've recorded your completion of Python functions with 80/100."`

1. **Check Progress** (reads `user:` state):

```
User: "What have I learned so far?"

Agent: [calls get_user_progress]
"You've completed 1 topic (Python functions) with an average score of 80."
```

1. **Search Past Lessons** (memory integration):

```
User: "What did we cover about functions?"

Agent: [calls search_past_lessons('functions')]
"I found 1 past session: Python functions where you scored 80%."
```
```

Option 2: CLI

```
adk run personal_tutor
```

Understanding the Behavior

Events Tab Debugging

In `adk web`, the **Events** tab shows:

1. **State Changes**:
2. `user:language` → "en" (persisted)
3. `user:difficulty_level` → "intermediate" (persisted)
4. `current_topic` → "Python functions" (session only)
5. `temp:quiz_percentage` → 80.0 (discarded after)
6. **Tool Calls**:
7. `set_user_preferences(language="en", difficulty_level="intermediate")`
8. `start_learning_session(topic="Python functions")`
9. `calculate_quiz_grade(correct_answers=8, total_questions=10)`
10. `record_topic_completion(topic="Python functions", quiz_score=80)`
11. **Output Key**:
12. `last_tutor_response` → Contains agent's final teaching response

State Lifecycle

```
┌──────────────────────────────────────────────────────────┐
│ Session 1: User sets preferences                         │
│ ┌──────────────────────────────────────────────────────┐ │
│ │ user:language = "en"           (PERSISTENT)          │ │
│ │ user:difficulty_level = "intermediate"  (PERSISTENT) │ │
│ │ user:topics_covered = []       (PERSISTENT)          │ │
│ └──────────────────────────────────────────────────────┘ │
└──────────────────────────────────────────────────────────┘

┌──────────────────────────────────────────────────────────┐
│ Session 2: User learns Python functions                  │
│ ┌──────────────────────────────────────────────────────┐ │
│ │ [READS user: state from Session 1]                   │ │
│ │ current_topic = "Python functions"  (SESSION ONLY)   │ │
│ │ temp:quiz_percentage = 80.0  (INVOCATION ONLY)       │ │
│ │ user:topics_covered = ["Python functions"]  (UPDATE) │ │
│ │ user:quiz_scores = {"Python functions": 80}  (UPDATE)│ │
│ └──────────────────────────────────────────────────────┘ │
└──────────────────────────────────────────────────────────┘

┌──────────────────────────────────────────────────────────┐
│ Session 3: User returns later                            │
│ ┌──────────────────────────────────────────────────────┐ │
│ │ [READS user: state with previous progress]           │ │
│ │ user:topics_covered = ["Python functions"]  (AVAILABLE) │ │
│ │ user:quiz_scores = {"Python functions": 80}  (AVAILABLE)│ │
│ │ temp:quiz_percentage = ???  (GONE! Not persisted)    │ │
│ └──────────────────────────────────────────────────────┘ │
└──────────────────────────────────────────────────────────┘
```

How It Works: State Management Deep Dive

1. User Preferences (Persistent)

```
# Tool function
tool_context.state['user:language'] = 'en'  # Persistent across sessions
tool_context.state['user:difficulty_level'] = 'intermediate'

# Later access (different session, same user)
language = tool_context.state.get('user:language', 'en')  # Returns 'en'!
```

Why: `user:` prefix stores data tied to `user_id`, available in all future sessions.

2. Session State (Session-Scoped)

```
# Current session tracking
tool_context.state['current_topic'] = 'Python functions'  # No prefix = sessio

# New session starts
# current_topic is GONE (unless using persistent SessionService)
```

Why: No prefix = data lives only within this session (unless SessionService persists it).

3. Temporary State (Invocation-Scoped)

```
# During quiz calculation
tool_context.state['temp:quiz_percentage'] = 80.0  # Invocation only

# After invocation completes
# temp:quiz_percentage is GONE forever
```

Why: `temp:` is for intermediate calculations, never persisted, always discarded.

4. Output Key (Auto-Save Response)

```
root_agent = Agent(
    ...,
    output_key="last_tutor_response"  # Saves agent's final response
)

# After agent responds
state['last_tutor_response'] = "Here's what we learned..."  # Auto-saved!
```

Why: Convenient way to save agent's response without manual state updates.

5. Tool Context State Updates

```
def my_tool(tool_context: ToolContext):
    # All state changes are automatically tracked
    tool_context.state['key'] = 'value'
    # Framework creates EventActions.state_delta behind the scenes
    return {'status': 'success'}
```

Why: Using `tool_context.state` ensures changes are tracked in events and persisted correctly.

Memory Service Integration (Production)

Setup for Vertex AI Memory Bank

Prerequisites:

1. Google Cloud Project with Vertex AI API enabled
2. Agent Engine created in Vertex AI
3. Authentication: `gcloud auth application-default login`

4. Environment variables:
   `bash export GOOGLE_CLOUD_PROJECT="your-gcp-project-id" export GOOGLE_CLOUD_LOCATION="us-central1"`

Configuration:

```
# Option 1: CLI flag
adk web personal_tutor --memory_service_uri="agentengine://1234567890"

# Option 2: Programmatic (modify agent.py)
from google.adk.memory import VertexAiMemoryBankService
from google.adk.runners import Runner

memory_service = VertexAiMemoryBankService(
    project="your-project-id",
    location="us-central1",
    agent_engine_id="1234567890"
)

runner = Runner(
    agent=root_agent,
    app_name="personal_tutor",
    memory_service=memory_service
)
```

Using Memory Tools:

```
from google.adk.tools.preload_memory_tool import PreloadMemoryTool
from google.adk.tools.load_memory_tool import LoadMemoryTool

root_agent = Agent(
    ...,
    tools=[
        PreloadMemoryTool(),  # Always loads memory at start
        # OR
        LoadMemoryTool(),  # Loads when agent decides
        # ... your other tools
    ]
)
```

Saving Sessions to Memory:

```
# Manual approach
await memory_service.add_session_to_memory(session)

# Automated with callback
async def save_to_memory_callback(callback_context):
    await callback_context.memory_service.add_session_to_memory(
        callback_context.session
    )

root_agent = Agent(
    ...,
    after_agent_callback=save_to_memory_callback
)
```

Key Takeaways

1. **State Prefixes Control Scope**:
2. No prefix → Session-level (depends on SessionService)
3. `user:` → Cross-session, user-specific (persistent)
4. `app:` → Cross-user, application-wide (persistent)
5. `temp:` → Invocation-only (always discarded)
6. **Update State via Context**:
7. Use `tool_context.state` or `callback_context.state`
8. **Never** directly modify `session.state` from `get_session()`
9. Changes are automatically tracked in EventActions
10. **Output Key Simplifies Response Saving**:
11. `output_key="key_name"` auto-saves agent's response
12. No manual state updates needed
13. **Memory Enables Long-Term Recall**:
14. `add_session_to_memory()` ingests conversations
15. `search_memory(query)` retrieves relevant past interactions
16. VertexAI Memory Bank provides semantic search

17. **Persistent Storage Requires Persistent SessionService**:
18. `InMemorySessionService` → Lost on restart
19. `DatabaseSessionService` / `VertexAiSessionService` → Persistent

Best Practices

State Management

DO:

- ✔️ Use `user:` for preferences that should persist
- ✔️ Use `temp:` for calculations that shouldn't persist
- ✔️ Use `tool_context.state` for updates
- ✔️ Use descriptive key names: `user:quiz_scores` not `scores`
- ✔️ Initialize state with defaults: `state.get('key', default)`

DON'T:

- ❌ Modify `session.state` from `get_session()` directly
- ❌ Store complex objects (functions, connections) in state
- ❌ Use `temp:` for data needed across invocations
- ❌ Forget to check if keys exist before reading

Memory Service

DO:

- ✔️ Call `add_session_to_memory()` after meaningful interactions
- ✔️ Use semantic queries: "What did we learn about X?"
- ✔️ Combine memory search with current state
- ✔️ Use VertexAI Memory Bank for production

DON'T:

- ❌ Save every trivial interaction to memory

- ❌ Rely on InMemoryMemoryService for production
- ❌ Forget to configure memory service URI
- ❌ Assume memory search is instant (it's an API call)

Common Issues & Troubleshooting

Issue 1: State Not Persisting Across Sessions

Problem: Set `user:language = "en"` but it's gone in next session

Solutions:

1. Check SessionService type:
   `python # InMemorySessionService = NO persistence # Use DatabaseSessionService or VertexAiSessionService`
2. Verify `user:` prefix is used
3. Ensure `append_event` is called (framework does this automatically)

Issue 2: `temp:` State Appears Empty

Problem: Set `temp:score` but it's not available later

Cause: `temp:` state is **intentionally discarded** after invocation

Solution: Use session state (no prefix) or `user:` prefix if needed later

Issue 3: Memory Search Returns Nothing

Problems & Solutions:

Using InMemoryMemoryService:

- Must call `add_session_to_memory()` first
- Only does keyword matching (not semantic)
- Use exact words from session

Using VertexAI Memory Bank:

- Ensure Agent Engine is created and ID is correct
- Check authentication: `gcloud auth application-default login`
- Verify environment variables are set
- Wait for indexing (not instant)

Issue 4: Tool Context State Changes Not Saving

Problem: `tool_context.state['key'] = value` doesn't persist

Solutions:

1. Tool must return (even empty dict)
2. Check if using correct context type (`ToolContext` not just dict)
3. Verify SessionService is configured in Runner
4. Use persistent SessionService for cross-session data

Real-World Applications

1. Personalized Education

- Track student progress across multiple subjects
- Adapt difficulty based on past performance
- Remember learning preferences (visual, auditory, etc.)
- Search past lessons when student asks questions

2. Customer Support Agent

- Remember customer preferences (language, communication style)
- Track issue history and resolutions
- Search past support tickets for context
- Use `temp:` for ticket validation workflows

3. Healthcare Assistant

- Store patient preferences securely (`user:` prefix)
- Track medication reminders across sessions
- Remember past symptoms and treatments
- Search medical history for diagnosis support

4. Personal Shopping Assistant

- Remember size preferences, style, budget (`user:` state)
- Track purchase history
- Use `temp:` for cart calculations
- Search past purchases for recommendations

Next Steps

🚀 **Tutorial 09: Callbacks & Guardrails** - Add safety controls and monitoring to your agents
🚀 **Tutorial 10: Evaluation & Testing** - Learn systematic testing of state management

Exercises:

1. Add a `reset_progress` tool that clears `user:` state
2. Implement `get_recommendations` that suggests topics based on history
3. Add `user:learning_goals` to track long-term objectives
4. Create a quiz generator that uses past performance to adjust difficulty

---

Further Reading

- Session State Documentation (https://google.github.io/adk-docs/sessions/state/)
- Memory Service Guide (https://google.github.io/adk-docs/sessions/memory/)

- Vertex AI Memory Bank (https://cloud.google.com/vertex-ai/generative-ai/docs/agent-engine/memory-bank/overview)
- Context Objects Reference (https://google.github.io/adk-docs/context/)

Congratulations! You now understand how to build agents with persistent memory and context-aware state management. This enables truly personalized, production-ready agents.

Generated on 2025-10-19 17:56:26 from 08_state_memory.md

Source: Google ADK Training Hub