🕌

📘 Understanding `create_text_output_item` and `create_text_delta`

🎯 Overview

Both `create_text_output_item` and `create_text_delta` are helper methods from the **MLflow `ResponsesAgent` base class** that create properly formatted response objects for the Databricks Agents Framework. They serve different purposes in streaming responses.

📦 `create_text_output_item`

Purpose

Creates a **complete, finalized text response item** that signals the response is done.

Method Signature

```
def create_text_output_item(
    text: str,
    id: str
) → dict
```

Parameters

Parameter	Type	Description
`text`	`str`	The complete response text to return to the user
`id`	`str`	A unique identifier for this output item (typically a UUID)

Returns

Returns a dictionary representing an **OutputItem** in Databricks Responses API format:

```
{
   "type": "message",
   "id": "unique-uuid-here",
   "role": "assistant",
   "content": [
      {
         "type": "output_text",
         "text": "The actual response text here..."
      }
   ]
}
```

When to Use

- ✅ When you have the **complete response** ready
- ✅ For **non-streaming** responses
- ✅ When signaling **end of response** generation
- ✅ In the `predict_stream()` method after agent workflow completes

Example Usage in Your Code

```
def predict_stream(self, request: ResponsesAgentRequest) → Generator[ResponsesAgentStreamEvent, None, None]:
   # ... agent execution ...

   if final_state and "messages" in final_state and final_state["messages"]:
      response_text = final_state["messages"]

      # Create complete output item
      yield ResponsesAgentStreamEvent(
         type="response.output_item.done",
         item=self.create_text_output_item(
            text=response_text,  # Complete response
            id=str(uuid4())      # Unique ID
```

```
        )
    )
```

Real Example

```
from uuid import uuid4

# Creating a complete response
output_item = self.create_text_output_item(
    text="# Shipping Policy\\n\\nWe offer free shipping for orders above $50
0...",
    id=str(uuid4())
)

print(output_item)
# Output:
# {
#     "type": "message",
#     "id": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
#     "role": "assistant",
#     "content": [
#         {
#             "type": "output_text",
#             "text": "# Shipping Policy\\n\\nWe offer free shipping for orders ab
ove $500..."
#         }
#     ]
# }
```

Structure Breakdown

```
{
    "type": "message",              # Type of output (always "message" for text)
    "id": "uuid-string",            # Unique identifier for tracking
    "role": "assistant",            # Who's speaking (always "assistant" for AI res
ponses)
    "content": [                    # Array of content items
```

```
        {
          "type": "output_text",  # Content type (text output)
          "text": "..."           # The actual response text
        }
    ]
}
```

🔄 `create_text_delta`

Purpose

Creates a **partial text chunk** for true streaming responses, allowing incremental text delivery as it's generated (token by token or chunk by chunk).

Method Signature

```
def create_text_delta(
    delta: str,
    item_id: str
) → dict
```

Parameters

Parameter	Type	Description
`delta`	`str`	A **small chunk** of text being streamed (could be a single token or word)
`item_id`	`str`	The **same ID** used across all chunks for this response (maintains continuity)

Returns

Returns a dictionary representing a **text delta event** for progressive streaming:

```
{
    "type": "response.output_item.text_delta",
    "item_id": "same-uuid-for-all-chunks",
    "delta": "chunk of text"
}
```

When to Use

- ✅ For **true token-by-token streaming** (like ChatGPT typing effect)
- ✅ When LLM generates text **progressively**
- ✅ For **real-time user feedback** during generation
- ✅ When you want to show **partial responses** as they're created

Example Usage (Hypothetical Streaming)

```
def predict_stream_with_tokens(self, request: ResponsesAgentRequest) →
Generator:
    # If we were doing token-by-token streaming
    item_id = str(uuid4())  # Same ID for all chunks

    # Stream tokens as they're generated
    for token in llm.stream("Generate response..."):
        yield ResponsesAgentStreamEvent(
            **self.create_text_delta(
                delta=token,      # Single token: "Hello", " ", "world", "!"
                item_id=item_id   # Same ID for entire response
            )
        )

    # Signal completion
    yield ResponsesAgentStreamEvent(
        type="response.output_item.done",
        item_id=item_id
    )
```

Real Example

```
from uuid import uuid4

item_id = str(uuid4())  # Generate once, use for all chunks

# First chunk
delta1 = self.create_text_delta(delta="Hello", item_id=item_id)
```

```
# {"type": "response.output_item.text_delta", "item_id": "...", "delta": "Hell
o"}

# Second chunk
delta2 = self.create_text_delta(delta=" world", item_id=item_id)
# {"type": "response.output_item.text_delta", "item_id": "...", "delta": " worl
d"}

# Third chunk
delta3 = self.create_text_delta(delta="!", item_id=item_id)
# {"type": "response.output_item.text_delta", "item_id": "...", "delta": "!"}

# User sees: "Hello" → "Hello world" → "Hello world!"
```

Structure Breakdown

```
{
   "type": "response.output_item.text_delta",  # Event type for streaming ch
unks
   "item_id": "uuid-string",                # Same ID across all chunks
   "delta": "partial text"                # The incremental chunk
}
```

🔍 Key Differences

Aspect	`create_text_output_item`	`create_text_delta`
Purpose	Complete, finalized response	Incremental streaming chunk
When Used	Response is fully generated	During generation (token-by-token)
Text Parameter	Full response text	Small chunk/token
Event Type	`response.output_item.done`	`response.output_item.text_delta`
ID Usage	Unique per response	Same ID across all chunks
User Experience	See complete text at once	See text appear progressively
Use Case	Your current implementation	True streaming (not used in your code)

🎭 Usage Patterns

Pattern 1: Complete Response (Your Current Code)

```
def predict_stream(self, request: ResponsesAgentRequest) → Generator:
    # Generate complete response
    response_text = agent.invoke(query)["messages"]

    # Yield complete response once
    yield ResponsesAgentStreamEvent(
        type="response.output_item.done",
        item=self.create_text_output_item(
            text=response_text,  # ✅ Complete text
            id=str(uuid4())
        )
    )
```

User sees:

```
[Loading...] → "# Shipping Policy\\n\\nWe offer free shipping..."
```

Result: Complete text appears instantly

Pattern 2: Token Streaming (Alternative, Not in Your Code)

```
def predict_stream_tokens(self, request: ResponsesAgentRequest) → Gene
rator:
    item_id = str(uuid4())  # Same ID for all chunks

    # Stream tokens as generated
    for chunk in llm.stream(query):
        if chunk.content:
            yield ResponsesAgentStreamEvent(
                **self.create_text_delta(
                    delta=chunk.content,  # ✅ Incremental chunks
                    item_id=item_id       # ✅ Same ID
                )
        )

    # Signal completion
    yield ResponsesAgentStreamEvent(
        type="response.output_item.done",
        item_id=item_id
    )
```

User sees:

```
# → # Shipping → # Shipping Policy → # Shipping Policy\\n\\nWe → ...
```

Result: Text appears progressively (typing effect)

🧪 Practical Examples

Example 1: Your Current Implementation

```
# After agent completes workflow
final_state = {
    'customer_query': 'What is your shipping policy?',
    'query_category': 'General',
    'query_sentiment': 'Neutral',
    'messages': '# Shipping Policy\\n\\nWe provide free shipping for orders a
bove $500.'
}

# Create complete output item
output = self.create_text_output_item(
    text=final_state["messages"],
    id=str(uuid4())
)

yield ResponsesAgentStreamEvent(
    type="response.output_item.done",
    item=output
)
```

Result: User receives complete, formatted response instantly

Example 2: Hypothetical True Streaming

```
# If you wanted token-by-token streaming
item_id = str(uuid4())
full_response = ""

# LLM generates tokens progressively
for token in ["#", " Shipping", " Policy", "\\n\\n", "We", " provide", "..."]:
    full_response += token

    # Stream each token
    yield ResponsesAgentStreamEvent(
        **self.create_text_delta(delta=token, item_id=item_id)
    )

    # User sees progressive update

# Signal completion
yield ResponsesAgentStreamEvent(
    type="response.output_item.done",
    item_id=item_id
)
```

Result: User sees text typing out in real-time

🤔 Why Your Code Uses `create_text_output_item` Not `create_text_delta`

Your Workflow:

1. ✅ Agent runs **complete workflow** (categorize → sentiment → response)
2. ✅ LLM generates **entire response** at once
3. ✅ You get **final_state["messages"]** with complete text
4. ✅ You yield **one complete item**

Result: `create_text_output_item` is perfect for this pattern

When You'd Use `create_text_delta`:

1. LLM streams **token by token**
2. You want **progressive display** (typing effect)
3. You need to **show progress** for long generations
4. You're implementing **chat-like interfaces**

Your agent doesn't need this because:

- ❌ LangGraph nodes generate complete responses
- ❌ You use `.invoke()` not `.stream()` for LLM calls
- ❌ RAG retrieval is instantaneous (not progressive)

🔧 Implementation in ResponsesAgent Base Class

Here's what these methods likely look like in the MLflow source (simplified):

```
class ResponsesAgent:
    def create_text_output_item(self, text: str, id: str) → dict:
        """Create a complete text output item."""
        return {
            "type": "message",
            "id": id,
            "role": "assistant",
            "content": [
                {
                    "type": "output_text",
                    "text": text
                }
            ]
        }

    def create_text_delta(self, delta: str, item_id: str) → dict:
        """Create a text delta for streaming."""
        return {
            "type": "response.output_item.text_delta",
            "item_id": item_id,
            "delta": delta
        }
```

📊 Visual Comparison

Using `create_text_output_item`

```
Time:    0s ──────────────────────────▶ 2s
Agent:   [Processing workflow...]
User:    [Loading indicator...]

Time:    2s
Response: ✅ Complete text appears
```

Timeline:

- 0-2s: Agent processes (categorize → sentiment → generate response)
- 2s: Complete response sent to user
- User experience: Instant complete text

Using `create_text_delta`

```
Time:    0s ─────▶ 0.5s ───▶ 1s ───▶ 1.5s ───▶ 2s
Agent:   [Token] [Token] [Token] [Token]  [Token]
User:    "Hello"  "world" "!"     "How"    "are"

Progressive display: Hello → Hello world → Hello world! → ...
```

Timeline:

- 0-2s: Tokens stream continuously
- User sees text appear progressively
- User experience: Real-time generation (ChatGPT-like)

🎯 Best Practices

For `create_text_output_item`:

✅ **Do:**

- Use for complete responses (your current pattern)
- Generate unique UUID for each response
- Validate text is not empty before creating
- Use when workflow completes before returning

❌ **Don't:**

- Use for partial responses
- Reuse the same ID across multiple responses
- Create empty text items (validate first)

For `create_text_delta`:

✅ **Do:**

- Use same `item_id` across all chunks for one response
- Send small, frequent chunks for smooth streaming
- Signal completion with `response.output_item.done` event
- Handle network interruptions gracefully

❌ **Don't:**

- Use different IDs for chunks of same response
- Send huge chunks (defeats purpose of streaming)
- Forget to send completion signal
- Use when complete response is already available

🔄 Migration Example: Complete to Streaming

If you wanted to convert your code to true streaming:

Current (Complete Response):

```
def predict_stream(self, request):
    # ... process request ...
    # Get complete response
    final_state = self.agent.invoke({"customer_query": msg})
    response_text = final_state["messages"]

    # Yield complete item
    yield ResponsesAgentStreamEvent(
        type="response.output_item.done",
        item=self.create_text_output_item(
            text=response_text,
            id=str(uuid4())
        )
    )
```

Converted (Token Streaming):

```
def predict_stream(self, request):
    # ... process request ...

    item_id = str(uuid4())  # Single ID for all chunks

    # Stream through agent with token streaming
    for event in self.agent.stream({"customer_query": msg}, stream_mode=
["messages"]):
        if isinstance(event, tuple) and event[0] == "messages":
            message = event[1][0]
            if isinstance(message, AIMessageChunk) and message.content:
                # Yield each token
                yield ResponsesAgentStreamEvent(
                    **self.create_text_delta(
                        delta=message.content,
                        item_id=item_id
                    )
                )

    # Signal completion
    yield ResponsesAgentStreamEvent(
        type="response.output_item.done",
        item_id=item_id
    )
```

Note: This would require changes to how your LLM calls are made (use streaming mode)

✅ Summary Table

Feature	`create_text_output_item`	`create_text_delta`
Returns	Complete message object	Partial delta event
Parameter Name	`text` (complete)	`delta` (chunk)
ID Behavior	Unique per response	Shared across chunks
Event Type	`response.output_item.done`	`response.output_item.text_delta`
Content Structure	Full message with content array	Simple delta string
Completion Signal	Implicit (item itself)	Explicit (separate event)
Used In Your Code	✅ Yes (main pattern)	❌ No (not needed)
Best For	Batch responses	Real-time streaming

💡 Key Takeaways

1. `create_text_output_item` = "Here's the complete response" ✅ **You use this**
2. `create_text_delta` = "Here's another piece of the response" ⚠️ **You don't need this**
3. Your current implementation is **correct** for your use case:
    - LangGraph workflow generates complete responses
    - No need for token-by-token streaming
    - Users get fast, complete answers
4. You'd only need `create_text_delta` if:
    - Implementing ChatGPT-like typing effect
    - Very long responses (minutes of generation)
    - User experience requires progressive feedback

Your code is optimal for the customer support router agent pattern! 🚀