Advanced Concepts for Agentcore Runtime

Overview

This section explores advanced capabilities of Amazon Bedrock AgentCore Runtime, including streaming responses, session management, large payload handling, and multi-modal content processing. These tutorials demonstrate how to build sophisticated, production-ready agents that can handle complex real-world scenarios with optimal performance and user experience.

Streaming Responses with AgentCore Runtime

Streaming responses allow a responsive user experiences for operations generating large content or requiring significant processing time.

When implementing streaming with AgentCore Runtime:

- **Use** `async def` for your entrypoint function
- **Use** `yield` to stream chunks as they become available
- **AgentCore SDK automatically handles** Server-Sent Events (SSE) format

- **Clients receive** `Content-Type: text/event-stream` responses

Example Streaming Implementation

```
from bedrock_agentcore.runtime impo edr
from strands import Agent, tool
from strands.models import BedrockModel

app = BedrockAgentCoreApp()

# Configure your agent with tools
@tool
def get_weather():
    """Get current weather"""
    return "sunny"

model = BedrockModel(model_id="us.anthropi
agent = Agent(model=model, tools=[get_weat

@app.entrypoint
async def streaming_agent(payload):
    """
    Streaming entrypoint - yields chunks a
    """
    user_input = payload.get("prompt")

    try:
        # Stream each chunk using async ge
        async for event in agent.stream_as
            if "data" in event:
                yield event["data"]
    except Exception as e:
        # Handle errors in streaming conte
        yield {"error": str(e), "type": "s
```

Configure Runtime Deployment

We use our **starter toolkit** to configure the AgentCore Runtime deployment with an entrypoint. We also configure the starter kit to

auto create the Amazon ECR repository on launch

```
from bedrock_agentcore_starter_toolkit import Runtime

agentcore_runtime = Runtime()

response = agentcore_runtime.configure(
    entrypoint="streaming_agent.py",
    execution_role=your_iam_role_arn,
    auto_create_ecr=True,
    requirements_file="requirements.txt"
)
```

Launch to AgentCore Runtime

Next, we launch the streaming agent to the AgentCore Runtime. This will create the Amazon ECR repository and the AgentCore Runtime

```
# Deploy the streaming agent
launch_result = agentcore_runtime.launch()
```

Invoke AgentCore Runtime using AgentCore SDK

We invoke the AgentCore Runtime with a payload and receive streaming responses

```
# Simple invocation with automatic session
invoke_response = agentcore_runtime.invoke({
    "prompt": "What's the weather like?"
})
```

Alternate Approach - Invoke AgentCore Runtime using boto3 for Advanced Control

Alternatively we can invoke **AgentCore Runtime** with any AWS SDK. Here we demonstrate invocation using Boto3.

```
import boto3
import json

agentcore_client = boto3.client('bedrock-a

response = agentcore_client.invoke_agent_r
    agentRuntimeArn=agent_arn,
    qualifier="DEFAULT",
    payload=json.dumps({"prompt": "Calcula
)

# Handle streaming response
if "text/event-stream" in response.get("co
    for line in response["response"].iter_
        if line and line.startswith(b"data
            chunk = line[6:].decode("utf-8
            print(f"Received: {chunk}")
```

Session and Context Management

Amazon Bedrock AgentCore Runtime provides isolated sessions for each user interaction, enabling agents to maintain context and state across multiple invocations while ensuring complete security isolation between different users. This example showcases how AgentCore Runtime handles sessions, maintains context across multiple invocations, and how agents can

access runtime information through the context object.

For demonstration purposes, we will use a Strands Agent that showcases these session management capabilities.

Session Isolation and Security

AgentCore Runtime provides **complete session isolation** through dedicated microVMs:

- **Dedicated Resources**: Each session runs in its own microVM with isolated CPU, memory, and filesystem
- **Security Boundaries**: Complete separation between user sessions prevents data contamination
- **Deterministic Cleanup**: After session completion, the microVM is terminated and memory is sanitized

Session Lifecycle

Sessions in AgentCore Runtime follow a specific lifecycle:

1. **Creation**: A new session is created on first invocation with a unique `runtimeSessionId`
2. **Active State**: Session processes requests and maintains context
3. **Idle State**: Session waits for next invocation while preserving context
4. **Termination**: Session ends due to:
    - Inactivity (15 minutes)
    - Maximum lifetime (8 hours)
    - Health check failures

Context Persistence

Within a session, AgentCore Runtime maintains:

- **Conversation History**: Previous interactions and responses
- **Application State**: Variables and objects created during execution
- **File System**: Any files created or modified during the session
- **Environment Variables**: Custom settings and configurations

Implementation of Session and Context Management

```
from strands import Agent, tool
from strands_tools import calculator # Imp
import argparse
import json
from bedrock_agentcore.runtime import Bedr
from strands.models import BedrockModel
import asyncio
from datetime import datetime

app = BedrockAgentCoreApp()

# Create a custom tool
@tool
def weather():
    """ Get weather """ # Dummy implementa
    return "sunny"

@tool
def get_time():
    """ Get current time """
    return datetime.now().strftime("%Y-%m-

model_id = "us.anthropic.claude-sonnet-4-2
model = BedrockModel(
    model_id=model_id,
)
agent = Agent(
    model=model,
    tools=[
        calculator, weather, get_time
```

```
    ],
    system_prompt="""
    You're a helpful assistant. You can do
    tell the weather, and provide the curr
    Always start by acknowledging the user
    """
)

def get_user_name(user_id):
    users = {
        "1": "Maira",
        "2": "Mani",
        "3": "Mark",
        "4": "Ishan",
        "5": "Dhawal"
    }
    return users[user_id]

@app.entrypoint
def strands_agent_bedrock_handling_context
    """
    AgentCore Runtime entrypoint that demo

    Args:
        payload: The input payload contain
        context: The runtime context objec

    Returns:
        str: The agent's response incorpor
    """
    user_input = payload.get("prompt")
    user_id = payload.get("user_id")
    user_name = get_user_name(user_id)

    # Access runtime context information
    print("=== Runtime Context Information
    print("User id:", user_id)
    print("User Name:", user_name)
    print("User input:", user_input)
    print("Runtime Session ID:", context.s
    print("Context Object Type:", type(con
    print("=== End Context Information ===

    # Create a personalized prompt that in
    prompt = f"""My name is {user_name}. H

    Additional context: This is session {c
    Please acknowledge my name and provide

```

```
        response = agent(prompt)
        return response.message['content'][0][
    
if __name__ == "__main__":
        app.run()
```

Accessing Context using Context Object Structure

The `context` parameter in your entrypoint function provides access to runtime information:

```
@app.entrypoint
def strands_agent_bedrock_handling_context(
    # Access session information
    session_id = context.session_id
    # Use context information in your agent
```

Payload Flexibility

The `payload` parameter allows flexible data passing:

```
# Example payload structures
payload = {
    "prompt": "User's question",
    "user_id": "1",
    "preferences": {...},
    "context_data": {...}
}
```

This enables rich, structured communication between clients and agents while maintaining the session context provided by the runtime.

Handling Large Multi-Modal Payloads

Amazon Bedrock AgentCore Runtime handles large payloads up to 100MB, including multi-modal content such as Excel files and images.

Why is this important?

Handling Large Multi-Modal Payloads is an important capability for Amazon Bedrock AgentCore Runtime because it enables agents to process and analyze complex, real-world data at scale. In modern business environments, data often comes in various formats and sizes, including large Excel files, high-resolution images, and other rich media. By supporting payloads up to 100MB and providing multi-modal processing capabilities, the AgentCore Runtime allows agents to extract insights from these diverse data sources. This is crucial for empowering agents to tackle sophisticated use cases, make informed decisions, and provide comprehensive analysis – ultimately delivering more value to end-users

Implementation of Handling Large Multi-Modal Payloads

Below code demonstrates how to create an entrypoint for your agent that can process both Excel files and images from large payloads:

```
@app.entrypoint
def multimodal_data_processor(payload, con
    """
    Process large multi-modal payloads con

    Args:
```

```
        payload: Contains prompt, excel_da
        context: Runtime context informati

    Returns:
        str: Analysis results from both da
    """
    prompt = payload.get("prompt", "Analyz
    excel_data = payload.get("excel_data",
    image_data = payload.get("image_data",

    print(f"=== Large Payload Processing =
    print(f"Session ID: {context.session_i

    if excel_data:
        print(f"Excel data size: {len(exce
    if image_data:
        print(f"Image data size: {len(imag
    print(f"Excel data {excel_data}")
    print(f"Image data {image_data}")
    print(f"=== Processing Started ===")
    # Decode base64 to bytes
    excel_bytes = base64.b64decode(excel_d
    # Decode base64 to bytes
    image_bytes = base64.b64decode(image_d

    # Enhanced prompt with data context
    enhanced_prompt = f"""{prompt}
    Please analyze both data sources and p
    """

    response = agent(
        [{
            "document": {
                "format": "xlsx",
                "name": "excel_data",
                "source": {
                    "bytes": excel_bytes
                }
            }
        },
        {
            "image": {
                "format": "png",
                "source": {
                    "bytes": image_bytes
                }
```

amit sharma

```
        )
        return response.message['content'][0][
    if __name__ == "__main__":
        app.run()
```

Try it out

At an AWS Event

If you are following the workshop via workshop studio, now go to JupyterLab in SageMaker Studio. In the JupyterLab UI navigate to `01-AgentCore-runtime/03-advanced-concepts`

Self-paced

Here are notebooks that let you try out the implementations:

Framework	Use Case	Description
Strands Agent	Streaming Responses	Implementing streaming responses using Amazon Bedrock AgentCore Runtime with real-time capabilities
Strands Agent	Understanding Runtime Context	Working with runtime context and session management, maintaining context across

Framework	Use Case	Description
		multiple invocations
Strands Agent	Handling Multi-Modal Payloads	Processing large payloads up to 100MB including multi-modal and images

Previous Next