Highlighting AWS SAM CLI: Local Serverless Development

The AWS Serverless Application Model Command Line Interface (AWS SAM CLI) is an essential tool for developing, testing, and debugging serverless applications locally before deploying them to AWS. It allows you to simulate the AWS Lambda and API Gateway environments on your local machine, significantly accelerating the development feedback loop and reducing cloud development costs.

This guide will demonstrate basic and advanced use cases of AWS SAM CLI, leveraging your **Advanced Track** local environment setup, particularly how it integrates with Docker and can simulate cloud services.

Reference: This guide builds upon the concepts and setup described in **Section 4.2. Core Technology Deep Dive** of the **Core Handbook** and the **Progressive Path Setup Guide Deep-Dive Addendum**, as well as the conceptual mention in **Cloud Migration + Terraform Snippets Deep-Dive Addendum (Section 9.3)**.

Basic Use Case: Local Lambda Function Development & Testing

Objective: To demonstrate how to define a simple AWS Lambda function and its API Gateway endpoint using SAM, and then invoke it locally to test its functionality.

Role in Platform: Enable rapid iteration and testing of lightweight, event-driven ETL functions or API endpoints that will eventually run as AWS Lambdas, complementing Spark's distributed processing.

Setup/Configuration (Local Environment - Advanced Track):

1. **Ensure Docker is running:** SAM CLI uses Docker to run Lambda functions locally.
2. **Install AWS SAM CLI:** Follow the official AWS documentation for installation on your operating system.
3. **Create a new SAM project directory:**
   ```
   mkdir lambda_sam_etl
   cd lambda_sam_etl
   sam init --runtime python3.9 --app-template hello-world --name MyLocalETLLambda
   ```
   Choose the Hello World Example template, Python 3.9 runtime. This creates a template.yaml (SAM template) and hello_world directory with app.py.
4. **Modify hello_world/app.py for a simple ETL concept:**
   ```
   # lambda_sam_etl/hello_world/app.py
   import json

   def lambda_handler(event, context):
   ```

```
    """
    A simple Lambda function to simulate a lightweight ETL step.
    It takes an input event (e.g., a mock transaction) and adds a processing timestamp.
    """
    print("Received event:", json.dumps(event, indent=2))

    try:
        # Assume event body is JSON string for API Gateway proxy integration
        if 'body' in event and isinstance(event['body'], str):
            input_data = json.loads(event['body'])
        else:
            input_data = event # Direct invocation

        # Simulate a lightweight transformation: add a processing timestamp
        if isinstance(input_data, dict):
            input_data['processed_timestamp'] =
json.dumps(context.get_remaining_time_in_millis() / 1000.0) # Dummy timestamp
            message = "Data processed successfully (local)."
        else:
            message = "Invalid input data format."
            input_data = {} # Ensure input_data is a dict even if invalid

        return {
            "statusCode": 200,
            "body": json.dumps({
                "message": message,
                "processed_data": input_data
            }),
            "headers": {
                "Content-Type": "application/json"
            }
        }
    except json.JSONDecodeError:
        print("Error: Invalid JSON in event body.")
        return {
            "statusCode": 400,
            "body": json.dumps({"message": "Invalid JSON input"}),
            "headers": {
                "Content-Type": "application/json"
            }
        }
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        return {
            "statusCode": 500,
            "body": json.dumps({"message": f"Internal server error: {e}"}),
            "headers": {
                "Content-Type": "application/json"
            }
        }
```

Steps to Exercise:

1. **Build the SAM application:**
   ```
   sam build
   ```
   This command compiles your Lambda code and dependencies into a .aws-sam/build directory, ready for local execution.
2. **Start the local API Gateway:**
   ```
   sam local start-api
   ```
   This command starts a local web server that emulates API Gateway. It will print the local endpoint URL (e.g., http://127.0.0.1:3000).
3. Send a request to the local API:
   Open a new terminal and use curl to send a POST request.
   ```
   curl -X POST -H "Content-Type: application/json" \
       -d '{"transaction_id": "LOC-001", "amount": 100.0, "currency": "USD"}' \
       http://127.0.0.1:3000/hello
   ```
   (Replace /hello with the path shown by sam local start-api if different).

Verification:

- **Console Output (sam local start-api terminal):** You will see logs from your Lambda function (print statements) showing the received event and processing.
- **curl Output:** The curl command should return a JSON response similar to:
  ```
  {"message": "Data processed successfully (local).", "processed_data": {"transaction_id": "LOC-001", "amount": 100.0, "currency": "USD", "processed_timestamp": "..."}}
  ```
  This confirms your Lambda executed locally and performed the simulated transformation.

Advanced Use Case 1: Integrating with LocalStack for Mock AWS Services

Objective: To test a Lambda function that interacts with other AWS services (like S3 for data storage) by integrating SAM CLI with LocalStack, providing a full local cloud simulation.
Role in Platform: Crucial for testing Lambda-based micro-ETL or data transformation jobs that read from or write to S3, without incurring cloud costs or requiring live AWS credentials.

Setup/Configuration:

1. **Ensure LocalStack is running:** From your main data platform docker-compose.yml, ensure localstack service is uncommented and running (or run it separately).
   - Example docker-compose.yml snippet for LocalStack:

     ```
     # In your main docker-compose.yml
     localstack:
       image: localstack/localstack:latest
       ports:
         - "4566:4566" # Standard LocalStack port
         - "4510-4559:4510-4559" # For services on dynamic ports (e.g., S3)
       environment:
         # Set services to start (e.g., 's3' or 's3,lambda,sqs')
         SERVICES: s3,lambda,apigateway
         DEBUG: 1
         # Add this to map localstack to host.docker.internal for SAM
         HOSTNAME_EXTERNAL: localhost
       healthcheck:
         test: ["CMD", "curl", "-f", "http://localhost:4566/health"]
         interval: 10s
         timeout: 5s
         retries: 5
     ```

2. **Install boto3:** In your Lambda's requirements.txt (e.g., lambda_sam_etl/hello_world/requirements.txt), add boto3. Run pip install -r requirements.txt locally, then sam build.
3. **Modify hello_world/app.py to interact with S3:**

   ```
   # lambda_sam_etl/hello_world/app.py (updated)
   import json
   import os
   import boto3

   # Configure S3 client to point to LocalStack
   # This environment variable will be passed during `sam local invoke`
   S3_ENDPOINT_URL = os.environ.get("S3_ENDPOINT_URL", None)
   s3_client = boto3.client('s3', endpoint_url=S3_ENDPOINT_URL) if S3_ENDPOINT_URL else boto3.client('s3')

   def lambda_handler(event, context):
       print("Received event:", json.dumps(event, indent=2))
       bucket_name = "my-local-data-bucket"
       object_key = f"processed-data/{context.aws_request_id}.json"

       try:
   ```

```
        input_data = {}
        if 'body' in event and isinstance(event['body'], str):
            input_data = json.loads(event['body'])
        else:
            input_data = event

        input_data['processed_timestamp'] =
json.dumps(context.get_remaining_time_in_millis() / 1000.0)

        # Create bucket if it doesn't exist
        try:
            s3_client.head_bucket(Bucket=bucket_name)
        except s3_client.exceptions.ClientError as e:
            if e.response['Error']['Code'] == '404':
                print(f"Bucket {bucket_name} does not exist, creating...")
                s3_client.create_bucket(Bucket=bucket_name)
                print(f"Bucket {bucket_name} created.")
            else:
                raise # Re-raise other errors

        # Write processed data to S3
        s3_client.put_object(
            Bucket=bucket_name,
            Key=object_key,
            Body=json.dumps(input_data)
        )
        print(f"Object written to s3://{bucket_name}/{object_key}")

        return {
            "statusCode": 200,
            "body": json.dumps({
                "message": f"Data processed and stored in S3://{bucket_name}/{object_key}",
                "processed_data": input_data
            }),
            "headers": {
                "Content-Type": "application/json"
            }
        }
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        return {
            "statusCode": 500,
            "body": json.dumps({"message": f"Internal server error: {e}"}),
    "headers": {
      "Content-Type": "application/json"
    }
  }
```

Steps to Exercise:

1. **Ensure LocalStack is running and healthy.**
2. **Build SAM app again:** sam build
3. **Invoke Lambda locally, pointing to LocalStack:**
   ```
   sam local invoke MyLocalETLLambda \
     --event-str '{"body": "{\"transaction_id\": \"S3-TEST-001\", \"amount\": 250.0}"}' \
     --env-vars env.json
   ```

   - Create an env.json file in lambda_sam_etl/ to configure the S3 endpoint for LocalStack:
     ```
     {
       "MyLocalETLLambda": {
         "S3_ENDPOINT_URL": "http://host.docker.internal:4566"
       }
     }
     ```

     Note: host.docker.internal allows the Lambda container (run by SAM) to connect to LocalStack running directly on your host machine (or in a separate Docker network). If LocalStack is in the *same* Docker network as SAM's build container, you might use the LocalStack service name (e.g., http://localstack:4566). host.docker.internal is more general for this local setup.

Verification:

- **SAM CLI Output:** The sam local invoke command will show the Lambda's logs, including the "Object written to s3://..." message.
- **LocalStack Logs:** Check the logs of your LocalStack container (docker compose logs localstack). You should see S3 PutObject and CreateBucket calls.
- **LocalStack S3:** Use the AWS CLI configured for LocalStack (or LocalStack's UI if available):
  ```
  aws --endpoint-url=http://localhost:4566 s3 ls s3://my-local-data-bucket/
  aws --endpoint-url=http://localhost:4566 s3 cp
  s3://my-local-data-bucket/processed-data/<object_key>.json -
  ```

  You should see the bucket and the JSON object created by your Lambda.

Advanced Use Case 2: Event-Driven Processing (Simulating Kafka Triggers)

Objective: To demonstrate how to test a Lambda function that is designed to be triggered by

a Kafka event, allowing local debugging of consumer logic for streaming data.
Role in Platform: Develop and test serverless functions that act as lightweight consumers of Kafka topics, performing real-time transformations or triggering subsequent actions, complementing or replacing parts of Spark streaming for simpler tasks.
Setup/Configuration:

1. **Ensure Kafka is running:** From your main data platform docker-compose.yml, ensure kafka and zookeeper services are running.
2. **Update hello_world/app.py for Kafka event processing:**

```
# lambda_sam_etl/hello_world/app.py (updated for Kafka event)
import json
import base64

def lambda_handler(event, context):
    """
    Lambda function to process a mock Kafka event.
    Assumes the event structure from a Kafka trigger (e.g., MSK as an event source).
    """
    print("Received Kafka event:", json.dumps(event, indent=2))

    try:
        records = event.get('records', {}).get('example.com:RawFinancialTransactions', []) #
Adjust topic name as needed
        processed_messages = []

        for record in records:
            # Kafka event value is base64 encoded
            decoded_value = base64.b64decode(record['value']).decode('utf-8')
            message_data = json.loads(decoded_value)

            # Add a processing timestamp or perform a simple transformation
            message_data['kafka_processed_at'] =
json.dumps(context.get_remaining_time_in_millis() / 1000.0) # Dummy timestamp
            message_data['source_topic'] = record['topic']
            message_data['kafka_offset'] = record['offset']

            processed_messages.append(message_data)
            print(f"Processed message from topic {record['topic']}: {message_data}")

        return {
            "statusCode": 200,
            "body": json.dumps({
                "message": f"Successfully processed {len(processed_messages)} Kafka
messages.",
                "processed_records": processed_messages
            }),
            "headers": {
                "Content-Type": "application/json"
            }
        }

    except Exception as e:
        print(f"Error processing Kafka event: {e}")
        return {
            "statusCode": 500,
            "body": json.dumps({"message": f"Internal server error: {e}"}),
            "headers": {
                "Content-Type": "application/json"
            }
        }
```

3. **Create a mock Kafka event file:** In your lambda_sam_etl/ directory, create kafka_event.json. This mimics the structure of an event that Lambda receives from an MSK (Managed Streaming for Kafka) trigger.

```
{
  "eventSource": "aws:kafka",
  "eventSourceArn":
"arn:aws:kafka:us-east-1:123456789012:cluster/MyMSKCluster/a1b2c3d4-5678-90ab-cd
ef-111111111111-2",
  "records": {
    "example.com:RawFinancialTransactions": [
      {
        "topic": "raw_financial_transactions",
        "partition": 0,
        "offset": 100,
        "timestamp": "2024-06-13T10:00:00.000Z",
        "timestampType": "CREATE_TIME",
        "value":
"eyJ0cmFuc2FjdGlvbl9pZCI6ICJLRlQtMDAxIiwgImFtb3VudCI6IDE1MC4wLCAiY3VycmVu
Y3kiOiAiVVNEIn0=",
        "key": "base64encodedkey"
      },
      {
        "topic": "raw_financial_transactions",
        "partition": 0,
        "offset": 101,
        "timestamp": "2024-06-13T10:01:00.000Z",
    "timestampType": "CREATE_TIME",
    "value":
"eyJ0cmFuc2FjdGlvbl9pZCI6ICJLRlQtMDAyIiwgImFtb3VudCI6IDIwMC4wLCAiY3VycmVu
Y3kiOiAiRVVSIn0=",
    "key": "base64encodedkey"
   }
  ],
  "example.com:RawInsuranceClaims": [
    {
      "topic": "raw_insurance_claims",
      "partition": 1,
      "offset": 50,
      "timestamp": "2024-06-13T10:05:00.000Z",
      "timestampType": "CREATE_TIME",
      "value":
"eyJjbGFpbV9pZCI6ICJMQUNfMDAxIiwgInBvbGljeV9udW1iZXIiOiAiUE9MLTExMSIsICJjb
GFpbV9hbW91bnQiOiAxMDAwLjAsICJjbGFpbV90eXBlIjogImF1dG8ifQ==",
      "key": "base64encodedkey"
    }
  ]
 }
}
```

Note: The value fields are base64 encoded JSON strings for {"transaction_id": "KFT-001", "amount": 150.0, "currency": "USD"} for financial and {"claim_id": "LAC_001", "policy_number": "POL-111", "claim_amount": 1000.0, "claim_type": "auto"} for insurance.

Steps to Exercise:

1. **Build SAM app:** sam build
2. **Invoke Lambda locally with Kafka event:**
   sam local invoke MyLocalETLLambda --event kafka_event.json

Verification:

- **SAM CLI Output:** The console will show the Lambda's logs, indicating that it successfully parsed and processed the Kafka messages from both financial and insurance topics. The processed_records in the response body will contain the transformed data. This demonstrates how you can test Kafka-triggered Lambdas locally with mock event payloads.

Advanced Use Case 3: Layered Lambda Functions for Shared Dependencies

Objective: To demonstrate how to create and use Lambda Layers to share common code and dependencies across multiple Lambda functions, reducing deployment package sizes and promoting code reuse.

Role in Platform: Efficiently manage libraries for multiple serverless ETL functions, ensuring consistency and simplified maintenance, especially for common data models or utility functions.

Setup/Configuration:

1. **Define a Lambda Layer in template.yaml:** Add a Resources section for the layer.

   ```
   # lambda_sam_etl/template.yaml (updated)
   AWSTemplateFormatVersion: '2010-09-09'
   Transform: AWS::Serverless-2016-10-31
   Description: >
     MyLocalETLLambda

     Sample SAM Template for MyLocalETLLambda.

   Globals:
     Function:
       Timeout: 30
       MemorySize: 128

   Resources:
     MyLocalETLLambda:
       Type: AWS::Serverless::Function
       Properties:
         CodeUri: hello_world/
         Handler: app.lambda_handler
         Runtime: python3.9
         Architectures:
           - x86_64
         Events:
           HelloWorld:
             Type: Api
             Properties:
               Path: /hello
               Method: post
         Layers:
           - !Ref CommonUtilsLayer # Reference the layer here

     # Define the Lambda Layer
     CommonUtilsLayer:
       Type: AWS::Serverless::LayerVersion
       Properties:
         LayerContentUri: common_layer/
   ```

```
      CompatibleRuntimes:
        - python3.9
      RetentionPolicy: Retain # Keep layer even if stack is deleted
```

2. **Create the shared common code:** Create a new directory lambda_sam_etl/common_layer/python/ (SAM expects python/ subfolder) and add a utility file, e.g., my_utils.py.

   ```
   mkdir -p lambda_sam_etl/common_layer/python
   ```

   Example lambda_sam_etl/common_layer/python/my_utils.py:

   ```
   # lambda_sam_etl/common_layer/python/my_utils.py
   def format_timestamp(timestamp_str):
       """Formats a timestamp string for consistency."""
       from datetime import datetime
       try:
           dt_obj = datetime.fromisoformat(timestamp_str.replace('Z', '+00:00'))
           return dt_obj.strftime("%Y-%m-%d %H:%M:%S UTC")
       except ValueError:
           return "Invalid Timestamp"

   def calculate_tax(amount, rate=0.05):
       """Calculates a simple tax."""
       return round(amount * rate, 2)
   ```

3. **Modify hello_world/app.py to use the layer:**

   ```
   # lambda_sam_etl/hello_world/app.py (updated to use layer)
   import json
   import base64
   import os
   import boto3
   from my_utils import format_timestamp, calculate_tax # Import from the layer

   S3_ENDPOINT_URL = os.environ.get("S3_ENDPOINT_URL", None)
   s3_client = boto3.client('s3', endpoint_url=S3_ENDPOINT_URL) if S3_ENDPOINT_URL else boto3.client('s3')

   def lambda_handler(event, context):
       print("Received event:", json.dumps(event, indent=2))

       try:
           # Assume event body is JSON string for API Gateway proxy integration
           if 'body' in event and isinstance(event['body'], str):
               input_data = json.loads(event['body'])
   ```

```
        else:
            input_data = event # Direct invocation

        # Use functions from the layer
        original_timestamp = input_data.get('timestamp', 'N/A')
        input_data['formatted_timestamp_from_layer'] =
format_timestamp(original_timestamp)

        original_amount = input_data.get('amount', 0.0)
        input_data['calculated_tax_from_layer'] = calculate_tax(original_amount)

        input_data['processed_timestamp'] =
json.dumps(context.get_remaining_time_in_millis() / 1000.0)

        # ... (S3 writing logic from Advanced Use Case 1 if desired, or remove for simple
test)

        return {
            "statusCode": 200,
            "body": json.dumps({
                "message": "Data processed with layer functions.",
                "processed_data": input_data
            }),
            "headers": {
                "Content-Type": "application/json"
            }
        }
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        return {
            "statusCode": 500,
            "body": json.dumps({"message": f"Internal server error: {e}"}),
            "headers": {
                "Content-Type": "application/json"
            }
        }
```

Steps to Exercise:

1. **Build SAM app (layers will be built):**
   ```
   sam build
   ```

2. **Start local API Gateway:**
   ```
   sam local start-api
   ```

3. **Send a request:**

   ```
   curl -X POST -H "Content-Type: application/json" \
     -d '{"transaction_id": "LAYER-001", "timestamp": "2024-06-13T15:00:00Z", "amount": 200.0}' \
     http://127.0.0.1:3000/hello
   ```

Verification:

- **Console Output:** The sam local start-api terminal logs and curl response will show the processed_data including formatted_timestamp_from_layer and calculated_tax_from_layer fields, populated by functions from your shared layer. This confirms the Lambda successfully accessed and used code from the defined layer.

This concludes the guide for AWS SAM CLI.