CustomAuthorizer

```
"""
This function authorizes an API Gateway request using an API key stored
in AWS Secrets Manager. The API key is cached in environment variables
to avoid unnecessary calls to AWS Secrets Manager. This function is
intended to be used as a custom authorizer for API Gateway.


"""

import os
import json
import boto3

from typing import Dict, Any

from aws_lambda_powertools import Logger, Tracer
from botocore.exceptions import ClientError


logger = Logger()
tracer = Tracer()

secrets_client = boto3.client('secretsmanager')


def fetch_secret_value() -> Dict[str, Any]:
    """
    Fetches the secret value from AWS Secrets Manager and caches it in
environment variables.

    Returns:
        A dictionary containing the secret value.
    """
    try:
        secret = secrets_client.get_secret_value(
            SecretId=os.environ['SECRET_ID'])
        os.environ['API_KEY'] = json.loads(secret['SecretString'])
['rand']
        logger.info(f"Fetched secret: {secret['SecretString']}")
        return secret
    except ClientError as e:
        logger.error(f"Unable to fetch secret: {e}")
        raise e


def lambda_handler(event: Dict[str, Any], context: Any) -> Dict[str,
Any]:
    """
    Authorizes an API Gateway request.

    Args:
        event: A dictionary containing the API Gateway event.
```

```
        context: The Lambda context object.

    Returns:
        A dictionary containing the authorization policy.
    """
    logger.info(f"Received event: {event}")

    if 'API_KEY' not in os.environ:
        fetch_secret_value()

    try:
        token = event['headers']['x-api-key']
        method_arn = event['methodArn']

        if token == os.environ['API_KEY']:
            principal_id = 'user'
            effect = 'Allow'
            resource = method_arn
        else:
            principal_id = 'user'
            effect = 'Deny'
            resource = method_arn

        policy = generate_policy(principal_id, effect, resource)
        logger.info(f"Generated policy: {policy}")
        return policy

    except Exception as e:
        logger.error(f"Error: {e}")
        raise e


def generate_policy(principal_id: str, effect: str, resource: str) ->
Dict[str, Any]:
    """
    Generates a policy for API Gateway authorization.

    Args:
        principal_id: The principal ID.
        effect: The effect of the policy (Allow/Deny).
        resource: The resource to authorize access to.

    Returns:
        A dictionary containing the authorization policy.
    """
    auth_response = {}
    auth_response['principalId'] = principal_id

    if effect and resource:
        policy_document = {}
        policy_document['Version'] = '2012-10-17'

        statement_one = {}
        statement_one['Action'] = 'execute-api:Invoke'
        statement_one['Effect'] = effect
        statement_one['Resource'] = resource
```

```
        policy_document['Statement'] = [statement_one]
        auth_response['policyDocument'] = policy_document

    return auth_response
```

DashboardFunction

```
"""
This module provides a Lambda function that generates a table of AWS
cost and usage data or a table of AWS System Manager (SSM) command
invocation statuses based on user input. The function is triggered by an
AWS CloudWatch dashboard widget.

Functions:
    - generate_table_cost(data): Generates an HTML table with AWS cost
and usage data.
    - generate_table_ssm(data): Generates an HTML table with SSM command
statuses.
    - convert_filter(filter_input, start_time, end_time): Formats start
and end times and adds input filter to a list.
    - get_command_statuses(start_time, end_time): Returns a dictionary
of counts for each SSM command invocation status type.
    - get_cost_usage(start_time, end_time): Returns an HTML table of AWS
cost and usage data grouped by service.
    - lambda_handler(event, context): Main Lambda function that
determines which function to run based on user input and returns the
result as an HTML table.

"""

import boto3
from datetime import datetime
import os

session = boto3.Session()
cost_explorer = session.client("ce")
document_name = os.environ['SSM_DOCUMENT_NAME']
project = os.environ['PROJECT_NAME']
ssm_client = boto3.client("ssm")


def generate_table_ssm(data):
    """
    Generate an HTML table with SSM command statuses.

    Args:
        data (dict): A dictionary containing SSM command statuses.

    Returns:
        str: An HTML table with SSM command statuses.

    Example:
        >>> data = {'Pending': 10, 'InProgress': 5, 'Success': 20,
'Cancelled': 0, 'Failed': 1, 'TimedOut': 0,
        ...         'DeliveryTimedOut': 0, 'ExecutionTimedOut': 0,
'Incomplete': 0, 'LimitExceeded': 0}
        >>> generate_table_ssm(data)
        '...
```

```
'
    """
    html = """


    """
    total = 0
    for status, count in data.items():
        total += count
        if status == "Failed" and count > 0:
            html += f""
        else:
            html += f""
    html += f""
    html += "
```

Status	Count
{status}	{count}
{status}	{count}
Total	**{total}**

```
"
    return html


def generate_table_cost(data):
    """
    Generate an HTML table with AWS cost usage.

    Args:
        data (dict): A dictionary containing AWS cost usage.

    Returns:
        str: An HTML table with AWS cost usage.

    Example:
        >>> data = {'AWS Lambda': '0.0000001', 'Amazon RDS':
'0.0000002', 'Amazon S3': '0.0000003'}
        >>> generate_table_cost(data)
        '...
'
    """
    html = """


    """
    total_cost = 0
    for service, cost in data.items():
        cost = float(cost)
        total_cost += cost
        html += f""
    html += f""
    html += "
```

Service	Cost

{service}	${'{:,.7f}'.format(cost)}
Total	**${'{:,.7f}'.format(total_cost)}**

```
"
    return html


def convert_filter(filter_input, start_time, end_time):
    """
    Convert the input filter into a format that can be used with AWS
Systems Manager ListCommands API.

    Args:
        filter_input (dict): A dictionary representing the filter to be
converted.
        start_time (int): The start time in milliseconds.
        end_time (int): The end time in milliseconds.

    Returns:
        list: A list of dictionaries representing the converted filter.

    """
    result = []
    # format the start and end time
    start_time = datetime.utcfromtimestamp(
        start_time/1000).strftime('%Y-%m-%dT%H:%M:%SZ')
    end_time = datetime.utcfromtimestamp(
        end_time/1000).strftime('%Y-%m-%dT%H:%M:%SZ')

    # add the start and end time to the result list
    result.append({"key": "InvokedAfter", "value": start_time})
    result.append({"key": "InvokedBefore", "value": end_time})
    result.append({"key": "DocumentName", "value": document_name})

    # add the input filter to the result list
    result.append(filter_input)

    return result


def get_command_statuses(start_time, end_time):
    """
    Get the number of AWS Systems Manager commands in each status during
the specified time period.

    Args:
        start_time (int): The start time in milliseconds.
        end_time (int): The end time in milliseconds.

    Returns:
        dict: A dictionary where the keys are command statuses and the
values are the number of commands in that status.

    """
    filters = [
        {"key": "Status", "value": "Pending"},
```

```
        {"key": "Status", "value": "InProgress"},
        {"key": "Status", "value": "Success"},
        {"key": "Status", "value": "Cancelled"},
        {"key": "Status", "value": "Failed"},
        {"key": "Status", "value": "TimedOut"},
        {"key": "Status", "value": "DeliveryTimedOut"},
        {"key": "Status", "value": "ExecutionTimedOut"},
        {"key": "Status", "value": "Incomplete"},
        {"key": "Status", "value": "LimitExceeded"},
    ]
    result = {}
    for filter in filters:
        filterValue = convert_filter(filter, start_time, end_time)
        params = {
            'Filters': filterValue
        }
        next_token = None
        while True:
            if next_token:
                params['NextToken'] = next_token
            try:
                response = ssm_client.list_commands(**params)
            except Exception as e:
                if str(e) == "An error occurred (ThrottlingException)
when calling the ListCommands operation (reached max retries: 4): Rate
exceeded":
                    return "Try Again"
                else:
                    raise e

            if "Commands" in response:
                if filter['value'] in result:
                    result[filter['value']] += len(response['Commands'])
                else:
                    result[filter['value']] = len(response['Commands'])
            if 'NextToken' not in response:
                break
            next_token = response['NextToken']
    return result


def get_cost_usage(start_time, end_time):
    """
    Get the cost and usage breakdown by AWS service during the specified
time period.

    Args:
        start_time (int): The start time in milliseconds.
        end_time (int): The end time in milliseconds.

    Returns:
        str: A string representing an HTML table with the cost and usage
breakdown by service.

    """
    start = datetime.utcfromtimestamp(start_time / 1000).strftime('%Y-
%m-%d')
```

```
    end = datetime.utcfromtimestamp(end_time / 1000).strftime('%Y-%m-
%d')
    if start >= end:
        return 'Please adjust the start and end dates to be greater than
1 day apart'

    # Create a session
    session = boto3.Session()

    # Connect to Cost Explorer
    cost_explorer = session.client("ce")

    try:
        # Get the cost and usage breakdown by service
        result = cost_explorer.get_cost_and_usage(
            TimePeriod={
                "Start": start,
                "End": end
            },
            Granularity="MONTHLY",
            Metrics=["BlendedCost"],
            GroupBy=[
                {
                    "Type": "DIMENSION",
                    "Key": "SERVICE"
                }
            ],
            Filter={
                "Tags": {
                    "Key": "Project",
                    "Values": [
                        project
                    ]
                }
            }
        )
    except Exception as e:
        if "Start date (and hour) should be before end date (and hour)"
in str(e):
            return 'Please adjust the start and end dates to be greater
than 1 day apart'
        raise e

    # Extract the cost and usage data
    data = []
    for time_period in result["ResultsByTime"]:
        data.extend(time_period["Groups"])

    # Prepare the data for the table
    cost_and_usage = {}
    for item in data:
        cost_and_usage[item["Keys"][0]
                       ] = item["Metrics"]["BlendedCost"]["Amount"]

    # Generate the table
    table = generate_table_cost(cost_and_usage)
```

```
    # Return the table
    return table


def lambda_handler(event, context):
    """
    AWS Lambda function handler.

    Args:
        event (dict): A dictionary representing the event that triggered
the Lambda function.
        context (object): An object representing the runtime context of
the Lambda function.

    Returns:
        str: A string representing an HTML table with the cost and usage
breakdown by service, or the number of AWS Systems Manager commands in
each status during the specified time period.

    """
    start_time = event['widgetContext']['timeRange']['start']
    end_time = event['widgetContext']['timeRange']['end']
    query = event['widgetContext']['params']['name']
    if query == "getCostandUsage":
        html = get_cost_usage(start_time, end_time)
    elif query == "getInvocationStatus":
        command_statuses = get_command_statuses(start_time, end_time)
        if command_statuses == "Try Again":
            html = "

Something went wrong: Please try again by using the refresh button.

"
        else:
            html = generate_table_ssm(command_statuses)
    else:
        html = "No data"
    return f'{html}'
```

ExpeDatCancelRunDocument

```
"""
This function is used to cancel a run command on instance using
boto3.client('ssm')

Functions:
    - lambda_handler: The main function of the Lambda function. This
function is triggered by an API Gateway endpoint. The function uses the
AWS Systems Manager (SSM) API to query the status of the SSM Agent on
the managed instances.
    -
"""


# initialize tracer, metrics and logger
import boto3
import botocore
import time
import datetime
import json
import os
import base64
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools import Logger
tracer = Tracer()
metrics = Metrics()
logger = Logger()

# initialize boto3 client and resource
dynamodb = boto3.resource("dynamodb")
dynamodb_table = os.environ.get('DYNAMODB_TABLE')
table = dynamodb.Table(dynamodb_table)
ssm_client = boto3.client("ssm")


@tracer.capture_method(capture_response=False)
def retry_with_backoff(function, *args, **kwargs):
    """
    Retry the given function with exponential backoff and jitter.

    Args:
        function (callable): The function to retry.
        *args: Positional arguments to pass to the function.
        **kwargs: Keyword arguments to pass to the function.

    Returns:
        The result of the function.

    Raises:
        botocore.exceptions.ClientError: If the maximum number of
```

```
retries is exceeded.
    Example:
        >>> retry_with_backoff(ssm_client.send_command,
InstanceIds=instance_ids, DocumentName=documeName,
Parameters=parameters)
        {'Command': {'CommandId': 'command_id', 'DocumentName': 'AWS-
RunShellScript', 'Comment': 'string', 'ExpiresAfter': datetime(2015, 1,
1), 'Parameters': {'commands': ['ls -l']}, 'InstanceIds':
['m-1234567890abcdef0'], 'Targets': [{'Key': 'tag:Name', 'Values':
['string']}], 'RequestedDateTime': datetime(2015, 1, 1), 'Status':
'Pending'|'InProgress'|'Success'|'Cancelled'|'Failed'|'TimedOut'|'Cancel
ling', 'StatusDetails': 'string', 'OutputS3Region': 'string',
'OutputS3BucketName': 'string', 'OutputS3KeyPrefix': 'string',
'MaxConcurrency': 'string', 'MaxErrors': 'string', 'TargetCount': 123,
'CompletedCount': 123, 'ErrorCount': 123, 'DeliveryTimedOutCount': 123,
'ServiceRole': 'string', 'NotificationConfig': {'NotificationArn':
'string', 'NotificationEvents':
['All'|'InProgress'|'Success'|'TimedOut'|'Cancelled'|'Failed'],
'NotificationType': 'Command'|'Invocation'}, 'CloudWatchOutputConfig':
{'CloudWatchLogGroupName': 'string', 'CloudWatchOutputEnabled': True|
False}}, 'ResponseMetadata': {'RequestId': 'request_id',
'HTTPStatusCode': 200, 'HTTPHeaders': {'x-amzn-requestid': 'request_id',
'content-type': 'application/x-amz-json-1.1', 'content-length': '1234',
'date': 'date'}, 'RetryAttempts': 0}}
    """
    max_retries = 5
    base_wait_time = 1
    retry_count = 0

    while True:
        try:
            return function(*args, **kwargs)
        except botocore.exceptions.ClientError as e:
            error_code = e.response["Error"]["Code"]

            if error_code in ["ThrottlingException",
"RequestLimitExceeded", "InternalFailure"]:
                retry_count += 1

                if retry_count > max_retries:
                    raise e

                wait_time = base_wait_time * 2 ** (retry_count - 1)
                time.sleep(wait_time)
            else:
                raise e


# function to fetch instance id from dynamodb table using theater_id and
return instance id for each theater
@tracer.capture_method(capture_response=False)
def get_instance_id(client_id):
    """
    Get the instance ID for a given client.

    Args:
        client_id (str): The ID of the client.
```

```
    Returns:
        dict or None: A dictionary containing the instance ID for the
given client ID, or None if an error occurred.
        str or None: An error message if applicable, or None if no
errors occurred.

    Raises:
        botocore.exceptions.ClientError: If there was an issue with the
client's request.
    Example:
        >>> get_instance_id("PVR-001", table)
        {"1234567890": "m-1234567890abcdef0"}, None
    """
    instance_id = {}  # dictionary with client_id as key and instance_id
as value
    try:
        response = table.get_item(Key={"TheaterId": client_id})
        if "Item" in response:
            tracer.put_annotation(
                key="client_id", value=response["Item"]["TheaterId"])
            tracer.put_annotation(
                key="instance_id", value=response["Item"]["InstanceId"])

            instance_id[client_id] = response["Item"]["InstanceId"]
        else:
            return None, f"Client ID {client_id} not found"
    except botocore.exceptions.ClientError as e:
        return None, json.dumps(e.response)
    return instance_id, None

# function to send command to instance using boto3.client('ssm') and
return job id and instance ids


@tracer.capture_method(capture_response=False)
def cancel_command(instance_id, command_id, max_retries=5, delay=5):
    """
    Cancels a command on a specified instance.

    Args:
        instance_id (str): The ID of the instance to send the command
to.
        command_id (str): The ID of the command to cancel.
        max_retries (int): The maximum number of times to retry sending
the command if throttling errors occur.
        delay (int): The delay in seconds between retry attempts, which
increases with each retry attempt.

    Returns:
        dict: A dictionary containing the job ID of the command, the
instance ID of the target instance,
        and the status of the command.
    Example:
        >>> cancel_command("m-1234567890abcdef0", "command_id")

    """
```

```
    for i in range(max_retries):
        try:
            result = ssm_client.cancel_command(
                InstanceIds=[instance_id], CommandId=command_id)
            logger.info(f"Command cancelled: {result}")
            return result

        except botocore.exceptions.ClientError as e:
            error_code = e.response['Error']['Code']
            if error_code == 'ThrottlingException':
                logger.warning(
                    f"Request throttled, retrying in {delay}
seconds...")
                time.sleep(delay)
                delay *= 2  # exponential backoff
            else:
                logger.error(f"Error sending command: {e.response}")
                return {"statusCode": 400, "body":
json.dumps(e.response)}
    logger.error(
        f"Max retries exceeded, unable to send command to instance
{instance_id}")
    return {"statusCode": 400, "body": json.dumps("Max retries
exceeded")}


# main lambda handler function
@tracer.capture_lambda_handler(capture_response=False)
@logger.inject_lambda_context
def lambda_handler(event: dict, context: LambdaContext):
    """
    Lambda handler function acts as the entry point.

    Args:
        event (dict): The event data passed to the function.
        context (LambdaContext): The context object passed to the
function.

    Returns:
        dict: A dictionary containing the response status code and body.

    Raises:
        Exception: If an unexpected error occurs.
    Example:
        >>> lambda_handler({"body": {"ClientId": "1234567890", "JobId":
"job_id"}}, {})
    """
    body = json.loads(event["body"])
    client_id = body["ClientId"]
    job_id = body["JobId"]

    instance_id, error = get_instance_id(client_id)
    if error:
        return {"statusCode": 400, "body": json.dumps(error)}
```

```
    result = cancel_command(instance_id[client_id], job_id)
    return {"statusCode": 200, "body": json.dumps(result)}
```

ExpeDatSSMInvocationQuery

```
"""
This file contains the error codes and error messages for ExpeDat .
For more information, see the ExpeDat documentation at https://
www.dataexpedition.com/expedat/Docs/movedat/exit-codes.html .

"""
error_dict = {
    0: "Success: all requested operations completed successfully",
    1: "Could not determine the nature of the error",
    4: "An unsupported feature was requested",
    5: "Invalid address",
    8: "Object too large to be delivered",
    9: "Object unavailable",
    10: "Bad Credentials (username or password)",
    11: "Object is busy or locked - try again later",
    13: "Operation timed out",
    14: "A requested condition was not met",
    17: "Unsupported application version",
    18: "Invalid Argument",
    20: "Transaction lost or unrecognized",
    21: "Encryption required",
    22: "Requested encryption is not supported",
    23: "Requested key is not valid or not supported",
    24: "Request denied by configuration or user input",
    25: "Invalid pathname",
    26: "A server name is invalid or no server could be reached",
    27: "Insufficient privileges for requested action",
    28: "Requested feature not supported",
    29: "Operating system error",
    30: "Server capacity has been exceeded",
    31: "Exceeded resource limit based on credentials",
    32: "Session aborted",
    33: "Out of memory",
    34: "Directory already exists or requested state already set",
    36: "A partial upload was detected and resumption was not selected",
    37: "A partial upload was detected, but the file or meta data
appears to be corrupted",
    38: "A partial upload was detected, but the source and destination
files appear to be different",
    43: "A feature was requested which is not supported by the software
license",
    66: "Error in network system call",
    69: "Network buffer overflow",
    71: "Request expired because network connectivity was lost",
    72: "Address is not valid",
    74: "Port is not valid (out of range)",
    76: "ICMP Network is down",
    77: "ICMP Host is down",
    78: "ICMP No application on given port",
    80: "ICMP Network unknown",
    81: "ICMP Host unknown",
    82: "ICMP Net/Host/Filter Prohibited",
```

```
    248: "Problem with command line or configuration file",
    249: "Error accessing a local file",
    250: "An error occurred with the DEI toolkit",
    251: "An error occurred with the SEQ module",
    252: "An error occurred with the DOC module",
    253: "An error occurred with the MTP module",
    254: "License/registration problem",
    255: "Multiple actions were requested and some failed"
}


def get_error_message(error_message):
    # "failed to run commands: exit status 255"
    # extract the exit code from the error message string
    exit_code = None
    try:
        if error_message and isinstance(error_message, str):
            exit_code_start_index = error_message.find(
                'exit status ') + len('exit status ')
            if exit_code_start_index >= len('exit status '):
                exit_code_str =
error_message[exit_code_start_index:].split()[
                    0]
                if exit_code_str.isdigit():
                    exit_code = int(exit_code_str)
    except Exception as e:
        print(f"Exception in get_error_message: {e}")
    if exit_code is not None:
        return error_dict.get(exit_code, "Could not determine the nature
of the error")
    else:
        return error_message
```

ExpeDatSSMInvocationQuery

```
"""
Lambda function for the ExpeDatSSMInvocationQuery function. This
function is used to query the status of the ExpeDat DCP transfer on the
Theater Box's. The function is triggered by an API Gateway endpoint. The
function uses the AWS Systems Manager (SSM) API to query the status of
the SSM Agent on the managed instances.

Functions:
    - lambda_handler: The main function of the Lambda function. This
function is triggered by an API Gateway endpoint. The function uses the
AWS Systems Manager (SSM) API to query the status of the SSM Agent on
the managed instances.
    - extract_info: Extracts the info from the output of the command
invocation.
    - retry_with_backoff: Retries a function with exponential backoff.
    - get_instance_id: Gets the instance id of the managed instance.
    - get_connection_status: Gets the connection status of the theater
box.


"""

import boto3
import botocore
import json
import os
import re
import time
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools import Logger
from progress import extract_info_progress as extract_info_progress
from exit_codes import get_error_message
tracer = Tracer()
metrics = Metrics()

logger = Logger()

ssm_client = boto3.client("ssm")
dynamodb = boto3.resource("dynamodb")
tableName = os.environ.get('DYNAMODB_TABLE')
table = dynamodb.Table(tableName)

# helper function


def format_time(time_sec):
    if time_sec < 60:
        return '{:.1f} seconds'.format(time_sec)
    elif time_sec < 3600:
```

```
        min = int(time_sec // 60)
        sec = int(time_sec % 60)
        return '{} minutes {} seconds'.format(min, sec)
    else:
        hour = int(time_sec // 3600)
        min = int((time_sec % 3600) // 60)
        sec = int((time_sec % 3600) % 60)
        return '{} hours {} minutes {} seconds'.format(hour, min, sec)


def extract_info(log):
    try:
        pattern_size = r'Total size of files \(in ([A-Za-z]+)\):
([0-9.]+)'
        pattern_num_files = r'Total number of files: (.+)'
        pattern_time_taken = r'Total time taken \(in seconds\): (.+)'
        pattern_speed = r'Best speed \(in Mbps\): (.+)'
        pattern_avg_speed = r'Average speed \(in Mbps\): (.+)'

        size_match = re.search(pattern_size, log)
        if size_match is None:
            raise ValueError('Invalid log format: cannot extract file
size')
        num_files_match = re.search(pattern_num_files, log)
        if num_files_match is None:
            raise ValueError(
                'Invalid log format: cannot extract number of files')

        time_taken_match = re.search(pattern_time_taken, log)
        if time_taken_match is None:
            raise ValueError('Invalid log format: cannot extract time
taken')
        speed_match = re.search(pattern_speed, log)
        if speed_match is None:
            raise ValueError('Invalid log format: cannot extract best
speed')
        avg_speed_match = re.search(pattern_avg_speed, log)
        if avg_speed_match is None:
            raise ValueError(
                'Invalid log format: cannot extract average speed')

        size_unit = size_match.group(1)
        size_val = float(size_match.group(2))
        if size_unit.lower() == 'kb':
            size_val /= 1024**2
        elif size_unit.lower() == 'mb':
            size_val /= 1024
        elif size_unit.lower() == 'gb':
            pass
        else:
            return None

        num_files = int(num_files_match.group(1))
        time_taken = float(time_taken_match.group(1))
        speed = float(speed_match.group(1))
        avg_speed = float(avg_speed_match.group(1))
```

```
        return size_val, num_files, time_taken, speed, avg_speed
    except Exception as e:
        logger.error(e)
        return None


@tracer.capture_method(capture_response=False)
def retry_with_backoff(function, *args, **kwargs):
    """
    Retries a function with exponential backoff.

    Args:
        function (function): The function to retry.
        *args: The arguments to pass to the function.
        **kwargs: The keyword arguments to pass to the function.

    Returns:
        The return value of the function.
    Example:
        >>> retry_with_backoff(my_function, "arg1", "arg2",
kwarg1="kwarg1")

    """

    max_retries = 5
    base_wait_time = 1
    retry_count = 0

    while True:
        try:
            return function(*args, **kwargs)
        except botocore.exceptions.ClientError as e:
            error_code = e.response["Error"]["Code"]

            if error_code in ["ThrottlingException",
"RequestLimitExceeded", "InternalFailure"]:
                retry_count += 1

                if retry_count > max_retries:
                    raise e

                wait_time = base_wait_time * 2 ** (retry_count - 1)
                time.sleep(wait_time)
            else:
                raise e


@tracer.capture_method(capture_response=False)
def run_progress_command(instanceID):
    """
    Runs the progress command on the managed instance.

    Args:
        instanceID (str): The ID of the managed instance.

    Returns:
        str: The output of the command invocation.
```

```
    Raises:
        botocore.exceptions.ClientError: If there was an issue with the
client's request.

    Example:
        >>> run_progress_command("m-1234567890232")

    """
    commandID = None
    try:
        response = ssm_client.send_command(
            InstanceIds=[instanceID],
            DocumentName="AWS-RunShellScript",
            Parameters={"commands": ["tail -n 4 /tmp/movedat.log"]},
            TimeoutSeconds=60,
        )
        tracer.put_annotation("instance_id", instanceID)
        tracer.put_annotation("command_id", response["Command"]
["CommandId"])
        commandID = response["Command"]["CommandId"]
        logger.info(f"Command ID: {commandID}")
    except botocore.exceptions.ClientError as e:
        logger.error(e)
        return None
    # wait for the command to start and then get the output
    output = None
    time.sleep(1)
    while output is None:
        try:
            response = ssm_client.get_command_invocation(
                CommandId=commandID, InstanceId=instanceID
            )
            logger.info(f"Command Status:
{response['StandardOutputContent']}")
            tracer.put_annotation("instance_id", instanceID)
            tracer.put_annotation("command_id", commandID)
            tracer.put_annotation("status", response["Status"])
            tracer.put_annotation("output",
response["StandardOutputContent"])
            duration_str, total_size_str, downloaded_size_str,
estimated_time_str = extract_info_progress(
                response["StandardOutputContent"])
            return duration_str, total_size_str, downloaded_size_str,
estimated_time_str
        except botocore.exceptions.ClientError as e:
            logger.error(e)
            return None


@tracer.capture_method(capture_response=False)
def get_instance_id(theater_id):
    """
    Get the instance ID for a given client.

    Args:
        client_id (str): The ID of the client.
```

```
    Returns:
        str: The instance ID of the client.
    Raises:
        botocore.exceptions.ClientError: If there was an issue with the
client's request.
    Example:
        >>> get_instance_id("PVR-001")
        "m-1234567890232"
    """

    try:
        response = table.get_item(Key={"TheaterId": theater_id})
        tracer.put_annotation("theater_id", theater_id)
        tracer.put_annotation("instance_id", response["Item"]
["InstanceId"])
        return response["Item"]["InstanceId"]
    except botocore.exceptions.ClientError as e:
        logger.error(e)
        return None
    except KeyError:
        logger.warning(f"No item found for theater_id {theater_id}")
        return None


@tracer.capture_method(capture_response=False)
def get_connection_status(instance_id):
    """
    Get the connection status of an Amazon managed instance.

    This function retrieves the connection status for a given instance
by using the AWS Systems Manager API.

    Args:
        instance_id (str): The ID of the instance.

    Returns:
        str: The connection status of the instance.

    Example:
        >>> get_connection_status("m-1234567890232")
        "Online"

    Exeption:
        botocore.exceptions.ClientError: If there was an issue with the
client's request.


    """

    res = ssm_client.describe_instance_information(
        Filters=[{"Key": "InstanceIds", "Values": [instance_id]}]
    )
    client_status = res["InstanceInformationList"][0]["PingStatus"]

    tracer.put_annotation("instance_id", instance_id)
    tracer.put_annotation(
        "status", res["InstanceInformationList"][0]["PingStatus"])
```

```
    return client_status


@logger.inject_lambda_context
@tracer.capture_lambda_handler
def lambda_handler(event: dict, context: LambdaContext):
    """
    Lambda handler for the ExpeDatSSMInvocationQuery function.

    Args:
        event (dict): The event data passed to the function.
        context (LambdaContext): The context data passed to the
function.

    Returns:
        dict: A dictionary containing the response data.

    Raises:
        None

    Example:
        >>> lambda_handler({"JobId": "1234567890", "ClientId":
"PVR-001"}, {})
        {"statusCode": 200, "body": json.dumps({"ClientId": "PVR-001",
"JobId": "1234567890", "InstanceId": "m-1234567890abcdef0",
"ExecutionStartDateTime": "2020-01-01T00:00:00Z",
"ExecutionElapsedTime": "PT0S", "Status": "Success", "StatusDetails":
"Success", "StandardOutputContent": "Success", "StandardErrorContent":
""})}
    """

    # get command_id and theater_id from post call body
    body = json.loads(event["body"])
    command_id = body["JobId"]
    theater_id = body["ClientId"]

    # get instance_id from dynamodb table
    def get_instance_id_with_retry(theater_id):
        return retry_with_backoff(get_instance_id, theater_id)
    instance_id = get_instance_id_with_retry(theater_id)

    if not instance_id:
        logger.append_keys(theater_id=theater_id)
        logger.error("instance_id is incorrect")

        return {"statusCode": 400, "body": json.dumps({
            "ClientStatus": "ClientNotFound",
            "ClientId": theater_id,
        })}

    # capture instance status, plugin name from ssm
describe_instance_information
    def get_client_status_with_retry(instance_id):
        return retry_with_backoff(get_connection_status, instance_id)
    client_status = get_client_status_with_retry(instance_id)
    # check if instance is connected
```

```
    if client_status != "Online":
        logger.append_keys(instance_id=instance_id,
theater_id=theater_id)
        logger.error("instance is not online")
        return {"statusCode": 400, "body": json.dumps({
            "ClientStatus": "Offline",
            "ClientId": theater_id,
        })}

    def get_command_invocation_with_retry(command_id, instance_id):
        return retry_with_backoff(ssm_client.get_command_invocation,
command_id, instance_id)

    def get_command_invocation_with_retry(command_id, instance_id):
        return retry_with_backoff(ssm_client.get_command_invocation,
CommandId=command_id, InstanceId=instance_id)
    try:
        res = get_command_invocation_with_retry(command_id, instance_id)

        logger.append_keys(command_id=command_id,
                           instance_id=instance_id,
theater_id=theater_id)
        logger.info(
            f"Succesfully got command invocation for theater ID:
{theater_id}")
    except botocore.exceptions.ClientError as e:
        logger.error(e)
        return {"statusCode": 200, "body": json.dumps({
            "ClientStatus": client_status,
            "ClientId": theater_id,
            "Status": "JobId is incorrect",
        })}

    info = {}
    if res["StandardOutputContent"] != "" and
res["StandardOutputContent"] != " ":
        data = extract_info(res["StandardOutputContent"])
        if data:
            info = "Total size of files: {:.2f} GB, Total number of
files: {}, Total time taken: {}, Best speed: {:.2f} Mbps, Average speed:
{:.2f} Mbps".format(
                data[0], data[1], format_time(data[2]), data[3],
data[4])

        else:
            info = "No information available"
    # if status is in progress, call run_progress_command to get
progress
    # data = {}
    # if res["Status"] == "InProgress":
    #     duration_str, total_size_str, downloaded_size_str,
estimated_time_str = run_progress_command(
    #         instance_id)
    #     data = {
    #         "Duration": duration_str,
    #         "TotalSize": total_size_str,
    #         "DownloadedSize": downloaded_size_str,
```

```
    #         "EstimatedTime": estimated_time_str,
    #     }

    return {"statusCode": 200, "body": json.dumps({
        "ClientId": theater_id,
        "JobId": res.get("CommandId"),
        "InstanceId": res.get("InstanceId"),
        "ExecutionStartDateTime": res.get("ExecutionStartDateTime"),
        "ExecutionElapsedTime": res.get("ExecutionElapsedTime"),
        "ExecutionEndDateTime": res.get("ExecutionEndDateTime"),
        "Status": res.get("Status"),
        "StandardOutputContent": [
            {"rawOutput": res.get("StandardOutputContent")},
            {"info": info},
        ],
        "StandardErrorContent":
get_error_message(res.get("StandardErrorContent")),
        "ClientStatus": client_status,
        # "Progress": data,
    })}
```

ExpeDatSSMInvocationQuery

```
from datetime import datetime, timedelta


def extract_info_progress(input_str):
    lines = input_str.strip().split('\n')
    total_size = int(lines[-1].split('\t')[-1])
    downloaded_size = int(lines[-1].split('\t')[-2])
    duration = int(lines[-1].split('\t')[4])

    def convert_duration(duration):
        seconds = duration // 1000
        if seconds < 60:
            return f"{seconds}s"
        elif seconds < 3600:
            minutes = seconds // 60
            seconds %= 60
            return f"{minutes}m {seconds}s"
        else:
            hours = seconds // 3600
            seconds %= 3600
            minutes = seconds // 60
            seconds %= 60
            return f"{hours}h {minutes}m {seconds}s"

    def convert_size(size):
        suffixes = ['B', 'KB', 'MB', 'GB', 'TB']
        suffix_index = 0
        while size >= 1024 and suffix_index < len(suffixes) - 1:
            size /= 1024
            suffix_index += 1
        return f"{size:.2f} {suffixes[suffix_index]}"

    total_size_str = convert_size(total_size)
    downloaded_size_str = convert_size(downloaded_size)
    duration_str = convert_duration(duration)

    time_per_byte = duration / downloaded_size if downloaded_size > 0
else 0
    estimated_time = int((total_size - downloaded_size) *
                         time_per_byte) if time_per_byte > 0 else 0
    estimated_time_str = convert_duration(estimated_time)
    return duration_str, total_size_str, downloaded_size_str,
estimated_time_str
```

ExpeDatSSMRunDocumet

```
"""
This function is used to send command to instance using
boto3.client('ssm') and return job id and instance ids

Functions:
    - lambda_handler: The main function of the Lambda function. This
function is triggered by an API Gateway endpoint. The function uses the
AWS Systems Manager (SSM) API to query the status of the SSM Agent on
the managed instances.
    - retry_with_backoff: Retries a function with exponential backoff.
    - get_instance_id: Gets the instance id of the managed instance.
    - get_connection_status: Gets the connection status of the theater
box.
    - send_command : send command to instance using boto3.client('ssm')
and return job id and instance ids
"""


# initialize tracer, metrics and logger
import boto3
import botocore
import time
import datetime
import json
import os
import base64
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools import Logger
tracer = Tracer()
metrics = Metrics()
logger = Logger()

# initialize boto3 client and resource
dynamodb = boto3.resource("dynamodb")
dynamodb_table = os.environ.get('DYNAMODB_TABLE')
table = dynamodb.Table(dynamodb_table)
ssm_client = boto3.client("ssm")
s3 = boto3.client('s3')

# initialize environment variables
sns_topic = os.environ.get('SNS_TOPIC')
ssmServiceRoleArn = os.environ.get('SSM_SERVICE_ROLE_ARN')
documeName = os.environ.get('SSM_DOCUMENT_NAME')
hash_document = os.environ['HASH_DOCUMENT']
default_destination_path = os.environ.get('DEFAULT_PATH')
s3_bucket = os.environ.get('S3_BUCKET')
server_id = os.environ.get('SERVER_ID')
api_url = os.environ.get('API_GATEWAY_ENDPOINT')
date = datetime.datetime.now().strftime("%Y-%m-%d")
```

```
# use hash.py and convert it into basq64 string


def encode_file_to_base64(file_location):
    with open(file_location, "rb") as file:
        encoded_bytes = base64.b64encode(file.read())
        encoded_string = encoded_bytes.decode('utf-8')
    return encoded_string


hash_file = encode_file_to_base64('helpers/hash.py')
logparser_file = encode_file_to_base64('helpers/logparser.py')


def check_s3_file(bucket, key):
    # check if file exists in s3
    try:
        s3.head_object(Bucket=bucket, Key=key)
        return True
    except Exception as e:
        print(e)
        return False


@tracer.capture_method(capture_response=False)
def retry_with_backoff(function, *args, **kwargs):
    """
    Retry the given function with exponential backoff and jitter.

    Args:
        function (callable): The function to retry.
        *args: Positional arguments to pass to the function.
        **kwargs: Keyword arguments to pass to the function.

    Returns:
        The result of the function.

    Raises:
        botocore.exceptions.ClientError: If the maximum number of
retries is exceeded.
    Example:
        >>> retry_with_backoff(ssm_client.send_command,
InstanceIds=instance_ids, DocumentName=documeName,
Parameters=parameters)
        {'Command': {'CommandId': 'command_id', 'DocumentName': 'AWS-
RunShellScript', 'Comment': 'string', 'ExpiresAfter': datetime(2015, 1,
1), 'Parameters': {'commands': ['ls -l']}, 'InstanceIds':
['m-1234567890abcdef0'], 'Targets': [{'Key': 'tag:Name', 'Values':
['string']}], 'RequestedDateTime': datetime(2015, 1, 1), 'Status':
'Pending'|'InProgress'|'Success'|'Cancelled'|'Failed'|'TimedOut'|'Cancel
ling', 'StatusDetails': 'string', 'OutputS3Region': 'string',
'OutputS3BucketName': 'string', 'OutputS3KeyPrefix': 'string',
'MaxConcurrency': 'string', 'MaxErrors': 'string', 'TargetCount': 123,
'CompletedCount': 123, 'ErrorCount': 123, 'DeliveryTimedOutCount': 123,
'ServiceRole': 'string', 'NotificationConfig': {'NotificationArn':
'string', 'NotificationEvents':
```

```
['All'|'InProgress'|'Success'|'TimedOut'|'Cancelled'|'Failed'],
'NotificationType': 'Command'|'Invocation'}, 'CloudWatchOutputConfig':
{'CloudWatchLogGroupName': 'string', 'CloudWatchOutputEnabled': True|
False}}, 'ResponseMetadata': {'RequestId': 'request_id',
'HTTPStatusCode': 200, 'HTTPHeaders': {'x-amzn-requestid': 'request_id',
'content-type': 'application/x-amz-json-1.1', 'content-length': '1234',
'date': 'date'}, 'RetryAttempts': 0}}
    """
    max_retries = 5
    base_wait_time = 1
    retry_count = 0

    while True:
        try:
            return function(*args, **kwargs)
        except botocore.exceptions.ClientError as e:
            error_code = e.response["Error"]["Code"]

            if error_code in ["ThrottlingException",
"RequestLimitExceeded", "InternalFailure"]:
                retry_count += 1

                if retry_count > max_retries:
                    raise e

                wait_time = base_wait_time * 2 ** (retry_count - 1)
                time.sleep(wait_time)
            else:
                raise e


@tracer.capture_method(capture_response=False)
def get_last_two_folders(path):
    """
    Return the last two folders of the given path.

    Args:
        path (str): The path.

    Returns:
        str: The last two folders of the path.
    Example:
        >>> get_last_two_folders("/a/b/c/d/e")
        "/c/d/e"


    """
    path_components = path.split(os.path.sep)
    if len(path_components) == 3:
        # path has only two folders, so return the whole path
        return path
    else:
        # path has more than two folders, so return the last two
        folder = os.path.join('/', *path_components[-2:])
        return folder
```

```
# function to get connection status of instance and return online or
offline status
@tracer.capture_method(capture_response=False)
def get_connection_status(instance_id):
    """
    Get the connection status of a managed instance.

    This function retrieves the connection status for a given instance
by using the AWS Systems Manager API.

    Args:
        instance_id (str): The ID of the instance.

    Returns:
        dict: A dictionary containing the connection status of the
instance. The keys are the instance IDs, and the values
        are the connection status strings. If an error occurs, the
dictionary will be empty.

        str: If an error occurs, a string containing the error message.

    Raises:
        None

    Example:
        >>> get_connection_status("m-1234567890abcdef0")
        {"m-1234567890abcdef0": "connected"}, None
    """

    # dictionary with instance_id as key and connection status as value
    connection_status = {}

    for instance in instance_id.values():
        try:
            res = ssm_client.get_connection_status(Target=instance)
            connection_status[instance] = res["Status"]
            tracer.put_annotation(key="connection_status",
value=res["Status"])
            tracer.put_annotation(key="instance_id", value=instance)

        except botocore.exceptions.ClientError as e:
            return None, json.dumps(e.response)
    return connection_status, None


# function to fetch instance id from dynamodb table using theater_id and
return instance id for each theater
@tracer.capture_method(capture_response=False)
def get_instance_id(client_id):
    """
    Get the instance ID for a given client.

    Args:
        client_id (str): The ID of the client.

    Returns:
        dict or None: A dictionary containing the instance ID for the
```

```
given client ID, or None if an error occurred.
        str or None: An error message if applicable, or None if no
errors occurred.

    Raises:
        botocore.exceptions.ClientError: If there was an issue with the
client's request.
    Example:
        >>> get_instance_id("PVR-001", table)
        {"1234567890": "m-1234567890abcdef0"}, None
    """
    instance_id = {}  # dictionary with client_id as key and instance_id
as value
    try:
        response = table.get_item(Key={"TheaterId": client_id})
        if "Item" in response:
            tracer.put_annotation(
                key="client_id", value=response["Item"]["TheaterId"])
            tracer.put_annotation(
                key="instance_id", value=response["Item"]["InstanceId"])

            instance_id[client_id] = response["Item"]["InstanceId"]
        else:
            return None, f"Client ID {client_id} not found"
    except botocore.exceptions.ClientError as e:
        return None, json.dumps(e.response)
    return instance_id, None

# function to send command to instance using boto3.client('ssm') and
return job id and instance ids


@tracer.capture_method(capture_response=False)
def send_command(instance_id, source_path, sns_topic, ssmServiceRoleArn,
documeName, client_id, max_retries=5, delay=5):
    """
    Sends a command to the specified instance, copying files from the
source_path to the instance's default
    destination path.

    Args:
        instance_id (str): The ID of the instance to send the command
to.
        source_path (str): The source path for the files to be copied.
        sns_topic (str): The SNS topic ARN.
        ssmServiceRoleArn (str): The ARN of the service role used to
send the command.
        documeName (str): The name of the SSM document used to execute
the command.
        client_id (str): The ID of the client.
        max_retries (int): The maximum number of times to retry sending
the command if throttling errors occur.
        delay (int): The delay in seconds between retry attempts, which
increases with each retry attempt.

    Returns:
        dict: A dictionary containing the job ID of the command, the
```

```
instance ID of the target instance,
        and the status of the command.
    Example:
        >>> send_command("i-1234567890abcdef0", "/mnt/efs/1234567890",
"arn:aws:sns:us-east-1:1234567890:my-topic", "arn:aws:iam::
1234567890:role/SSMServiceRole", "AWS-RunShellScript", "1234567890")
        {"job_id": "1234567890", "instance_id": "m-1234567890abcdef0",
"status": "InProgress"}
    """
    # only include last 2 folder path from source path
    source_folder = get_last_two_folders(source_path)
    # Split source_folder into individual folder names
    folder_names = source_folder.strip("/").split("/")
    # add the end of the source folder to the destination path
    dest_folder = default_destination_path + "/".join(folder_names)

    for i in range(max_retries):
        try:
            result = ssm_client.send_command(
                InstanceIds=[instance_id],
                DocumentName=documeName,
                DocumentVersion="$LATEST",
                ServiceRoleArn=ssmServiceRoleArn,
                NotificationConfig={
                    "NotificationArn": sns_topic,
                    "NotificationEvents": [
                        "All",
                    ],
                    "NotificationType": "Invocation",
                },
                TimeoutSeconds=3600 * 24 * 2,
                Comment=f"Copy from {source_folder} to {client_id}",
                Parameters={
                    "SourcePath": [source_path],
                    "DestinationPath": [dest_folder],
                    "ClientId": [client_id],
                    "APIGatewayUrl": [api_url],
                    "GenerateHash": [hash_file],
                    "LogParser": [logparser_file],
                    "Date": [date]
                },
            )
            logger.info(f"Command sent to instance {instance_id}")
            tracer.put_annotation(
                key="job_id", value=result["Command"]["CommandId"])
            tracer.put_annotation(
                key="instance_id", value=result["Command"]
["InstanceIds"][0])
            tracer.put_metadata(key="status", value=result)
            response = {
                "JobId": result["Command"]["CommandId"],
                "InstanceId": result["Command"]["InstanceIds"][0],
                "Status": result["Command"]["Status"],
            }
            return response
        except botocore.exceptions.ClientError as e:
            error_code = e.response['Error']['Code']
```

```
            if error_code == 'ThrottlingException':
                logger.warning(
                    f"Request throttled, retrying in {delay}
seconds...")
                time.sleep(delay)
                delay *= 2  # exponential backoff
            else:
                logger.error(f"Error sending command: {e.response}")
                return {"statusCode": 400, "body":
json.dumps(e.response)}
    logger.error(
        f"Max retries exceeded, unable to send command to instance
{instance_id}")
    return {"statusCode": 400, "body": json.dumps("Max retries
exceeded")}


def generate_hash_server(server_id, source_path, ssmServiceRoleArn,
hash_document):
    """
    Generates a hash for the server using SSM run command.

    Returns:
        str: A hash for the server.
    Example:
        >>> generate_hash_server()
        "1234567890abcdef0"
    """

    folder_name = os.path.basename(source_path.rstrip('/'))
    try:
        result = ssm_client.send_command(
            InstanceIds=[server_id],
            DocumentName=hash_document,
            DocumentVersion="$LATEST",
            ServiceRoleArn=ssmServiceRoleArn,

            TimeoutSeconds=3600,
            Comment=f"Generate hash file for {folder_name} on server",
            Parameters={
                "SourcePath": [source_path],
                "APIURL": [api_url],
                "GenerateHash": [hash_file],
                "Date": [date]
            },
        )
    except botocore.exceptions.ClientError as e:
        error_code = e.response['Error']['Code']
        if error_code == 'ThrottlingException':
            logger.warning(
                f"Request throttled, retrying in {delay} seconds...")
            time.sleep(delay)
            delay *= 2
        else:
            logger.error(f"Error sending command: {e.response}")
            return {"statusCode": 400, "body": json.dumps(e.response)}
```

```
# main lambda handler function
@tracer.capture_lambda_handler(capture_response=False)
@logger.inject_lambda_context
def lambda_handler(event: dict, context: LambdaContext):
    """
    Lambda handler function acts as the entry point.

    Args:
        event (dict): The event data passed to the function.
        context (LambdaContext): The context object passed to the
function.

    Returns:
        dict: A dictionary containing the response status code and body.

    Raises:
        Exception: If an unexpected error occurs.
    Example:
        >>> lambda_handler({"body": {"ClientId": "1234567890",
"SourcePath": "/mnt/efs/1234567890/2020", "DestinationPath": "/mnt/efs/
1234567890/2020"}}, {})
        {"statusCode": 200, "body": {"JobId": "1234567890",
"InstanceId": "m-1234567890abcdef0", "Status": "Pending"}}
    """
    body = json.loads(event["body"])
    client_id = body["ClientId"]
    source_path = body["SourcePath"]
    folder_name = os.path.basename(source_path.rstrip('/'))
    server_hash_key = 'FolderIntegrity/Server/{date}/{folder_name}/
source.csv'.format(
        date=date,
        folder_name=folder_name
    )
    # destination_path = body["DestinationPath"]
    # get instance_id from dynamodb table
    instance_id, error = get_instance_id(client_id)
    if error:
        logger.error(f"Error getting instance ID: {error}")
        return {"statusCode": 400, "body": error}
    if client_id not in instance_id:
        logger.warning(f"Client ID {client_id} not found")
        return {"statusCode": 200, "body": {"InstanceId": client_id,
"ClientStatus": "unidentified"}}
    response = {}
    # get online and offline instance
    connection_status, error = get_connection_status(instance_id)
    if error:
        logger.error(f"Error getting connection status: {error}")
        return {"statusCode": 400, "body": error}

    def send_command_with_retry(instance, source_path, sns_topic, role,
documeName, client_id):
        return retry_with_backoff(send_command, instance, source_path,
sns_topic, role, documeName, client_id)
    # check in s3 if the hash file exists
    if not check_s3_file(s3_bucket, server_hash_key):
```

```
        logger.info(f"Hash file for {folder_name} does not exist on
server")
        # generate hash file on server
        response = generate_hash_server(
            server_id, source_path, ssmServiceRoleArn, hash_document)
        logger.info(f"Hash file for {folder_name} generated on server")

    try:
        for instance in connection_status:
            if connection_status[instance] == "connected":
                response = send_command_with_retry(
                    instance, source_path, sns_topic, ssmServiceRoleArn,
documeName, client_id)

                logger.append_keys(client_id=client_id)
                logger.info(
                    f"Command sent to instance {instance} successfully")
            else:
                logger.warning(f"Instance {instance} is offline")
                response = {
                    "InstanceId": instance,
                    "ClientStatus": "Offline",
                }
    except botocore.exceptions.ClientError as e:
        logger.error(f"Error sending command: {e.response}")
        return {"statusCode": 400, "body": json.dumps(e.response)}
    return {"statusCode": 200, "body": json.dumps(response)}
```

helpers

```
import hashlib
import os
import csv
import requests
import sys
from urllib.parse import urlparse
import datetime

date = datetime.datetime.now().strftime("%Y-%m-%d-%H-%M-%S")

# Get command-line arguments
destination_path = sys.argv[1]
source_path = sys.argv[2]
client_id = sys.argv[3]
api_gateway_url = sys.argv[4]
datefolder = sys.argv[5]

hashfile = '/tmp/{date}.csv'.format(date=date)

folder_name = os.path.basename(destination_path.rstrip('/'))

try:

    # Create the output file
    with open(hashfile, mode='w', newline='') as csv_file:
        writer = csv.writer(csv_file)
        writer.writerow(['filename', 'hash'])

        # Loop through all the files in the directory and its
subdirectories
        for root, dirs, files in os.walk(destination_path):
            for filename in files:
                # Open the file and generate its hash
                with open(os.path.join(root, filename), 'rb') as file:
                    file_hash = hashlib.sha256()
                    while chunk := file.read(8192):
                        file_hash.update(chunk)

                # Write the filename and hash to the output file
                writer.writerow(
                    [os.path.join(root, filename),
file_hash.hexdigest()])

    # Upload the CSV file to S3 using a presigned URL generated from the
Lambda function
    with open(hashfile, 'rb') as file:
        response = requests.post(
            api_gateway_url + '/presignedurl', json={'client_id':
client_id, 'folder_name': folder_name, 'date': datefolder})
        response.raise_for_status()
        presigned_url = response.json()['presigned_url']
```

```
        response = requests.put(presigned_url, data=file.read(),
                                headers={'Content-Type': 'text/csv'})
        response.raise_for_status()

    # extract the prefix from the presigned url
    parsed_url = urlparse(presigned_url)
    file_path = parsed_url.path[1:]  # removing the leading '/'
    # print(file_path)
    # verify the hash by calling the lambda function
    if client_id != 'Server':
        res = requests.post(api_gateway_url + '/verify', json={
                            'client_id': client_id, 'folder_name':
folder_name, 'client_key': file_path, 'date': datefolder})
        res.raise_for_status()
        result = res.json()
        # print(result)  # {'matches': 0, 'non_matches': 1,
'server_key': 'FolderIntegrity/Server/2021-05-05/2021-05-05-11-00-00/
source.csv', 'client_key': 'FolderIntegrity/Client/
2021-05-05/2021-05-05-11-00-00/destination.csv'}
        print(f"Server Key: {result['server_key']}")
        print(f"Client Key: {result['client_key']}")

        print(
            f"Found {result['matches']} matching files and
{result['non_matches']} non-matching files")

        # Check for non-matching files and raise an exception if found
        if result['non_matches'] > 0:
            raise Exception(f"{result['non_matches']} files do not
match")
    else:
        print('Hash file generated on Server')


except Exception as e:
    print(f"Error: {e}")
    sys.exit(1)
finally:
    # Delete the output file
    try:
        os.remove(hashfile)
    except Exception as e:
        print(f"Error: Failed to delete {hashfile}: {e}")
```

helpers

```
import hashlib
import os
import csv
import re
import sys
import argparse
from datetime import datetime


def parse_time(time_str):
    time = re.search(r'(\d+(\.\d+)?)\s+(ms|sec|min|hrs)', time_str)
    if time:
        time = time.group(0)
        if time.endswith('ms'):
            time_ms = time.split()[0]
        elif time.endswith('sec'):
            time_ms = float(time.split()[0]) * 1000
        elif time.endswith('hrs'):
            time_ms = float(time.split()[0]) * 1000 * 60 * 60
        elif time.endswith('min'):
            time_ms = float(time.split()[0]) * 1000 * 60
        else:
            raise Exception('Unknown time unit')
        return float(time_ms) / 1000.0
    else:
        raise Exception('Could not parse time')


def parse_speed(speed_str):
    match = re.search(r'(\d+\.\d+)\s+mbit/s', speed_str)
    if match:
        return float(match.group(1))
    else:
        raise Exception('Could not parse speed')


def get_folder_size(folder_path):
    total_size = 0
    for dirpath, dirnames, filenames in os.walk(folder_path):
        for f in filenames:
            fp = os.path.join(dirpath, f)
            total_size += os.path.getsize(fp)
    return total_size


def convert_size(size):
    if size > 1024 * 1024 * 1024:
        return size / (1024 * 1024 * 1024), 'GB'
    elif size > 1024 * 1024:
        return size / (1024 * 1024), 'MB'
    elif size > 1024:
        return size / 1024, 'KB'
```

```
    else:
        return size, 'bytes'


parser = argparse.ArgumentParser()
parser.add_argument("logfile", help="path to log file")
args = parser.parse_args()

try:
    log_file = args.logfile
    num_files = 0
    total_size = 0
    total_time = 0
    total_speed = 0
    high_speed = 0
    errmsg = ''
    errcount = 0
    folder_path = ''
    found_first_I = False

    with open(log_file, 'r') as f:
        errmsg += '\n'
        for line in f:
            try:
                if line.startswith('I') and not found_first_I:
                    folder_path = os.path.dirname(line.split()[5])
                    source_path = os.path.dirname(line.split()[8])
                    found_first_I = True
                elif line.startswith('F'):
                    fields = line.split('\t')
                    if len(fields) < 11:
                        raise Exception('Invalid line format')
                    size_time_speed = fields[10]
                    file_size_str, rest = size_time_speed.split(' in ')
                    time_sec = parse_time(rest)
                    speed_mbps = parse_speed(size_time_speed)
                    total_speed += speed_mbps
                    total_time += time_sec
                    if speed_mbps > high_speed:
                        high_speed = speed_mbps
                elif line.startswith('W'):
                    errcount += 1
                    timestamp = line.split()[1] + ' ' + line.split()[2]
                    if errmsg:
                        errmsg += '\n'
                    errmsg += f"{timestamp}: {' '.join(line.split()
[3:])}"

                else:
                    continue
            except Exception as e:
                errcount += 1
                timestamp = line.split()[1] + ' ' + line.split()[2]
                if errmsg:
                    errmsg += '\n'
                errmsg += f"{timestamp}: {' '.join(line.split()[3:])}"
    # use folder_path to get total number of files
```

```
    num_files = len(os.listdir(folder_path))
    total_size = get_folder_size(folder_path)
    avg_speed = total_speed / num_files if num_files > 0 else 0
    total_size, size_unit = convert_size(total_size)

    print(f"Total number of files: {num_files}")
    print(f"Total size of files (in {size_unit}): {total_size}")
    print(f"Total time taken (in seconds): {total_time}")
    print(f"Best speed (in Mbps): {high_speed}")
    print(f"Average speed (in Mbps): {avg_speed}")
    print(f"Destination Path: {folder_path}")
    print(f"Source Path: {source_path}")
    print()
    print(f'Error count: {errcount}' if errcount > 0 else '')
    print(f"Error Logs: {errmsg}" if errcount > 0 else '')
except Exception as e:
    print("An error occurred: {}".format(str(e)))
    sys.exit(1)
```

GeneratedPresignedURL

```
import boto3
from botocore.exceptions import ClientError
import json
import os
import datetime

s3 = boto3.client('s3')
bucket_name = os.environ['S3_BUCKET']


def lambda_handler(event, context):
    # {'client_id': client_id, 'folder_name': folderName}
    # Get the bucket name from the request body
    print(event)

    body = json.loads(event['body'])
    prefix = 'Server'
    filename = 'source.csv'
    client_id = ''
    date = ''
    if body['date'] is not None:
        date = body['date']
    else:
        date = datetime.datetime.now().strftime("%Y-%m-%d")

    # if body['client_id'] is 'Server': # This is a server request
    if body['client_id'] == 'Server':  # This is a server request
        prefix = 'Server'
        filename = 'source.csv'
        s3Key = 'FolderIntegrity/{prefix}/{date}/{folder_name}/
{filename}'.format(
            prefix=prefix,
            date=date,
            folder_name=body['folder_name'],
            filename=filename
        )
    else:  # This is a client request
        prefix = 'Client'
        filename = 'destination.csv'
        client_id = body['client_id']
        s3Key = 'FolderIntegrity/{prefix}/{client_id}/{date}/
{folder_name}/{filename}'.format(
            prefix=prefix,
            client_id=client_id,
            date=date,
            folder_name=body['folder_name'],
            filename=filename
        )

    # Generate a presigned URL for uploading to S3
    try:
        presigned_url = s3.generate_presigned_url(
```

```
            'put_object',
            Params={'Bucket': bucket_name, 'Key': s3Key},
            ExpiresIn=3600,
        )
    except ClientError as e:
        print(e)
        return {
            'statusCode': 500,
            'body': 'Failed to generate presigned URL'
        }

    # Return the presigned URL in a JSON object
    return {
        'statusCode': 200,
        'body': json.dumps({'presigned_url': presigned_url})
    }
```

SSMAgentActivation

```
"""
This function creates activation code and id for SSM agent and stores it
in SSM parameter store. It also updates the activation code and id in
SSM parameter store if the activation code is not expired and the
registration limit is not reached. It also adds theater id as tags to
the instance.

Functions:
    - create_activation: This function creates activation code and id
for SSM agent and stores it in SSM parameter store.
    - update_ssm: This function updates the activation code and id in
SSM parameter store if the activation code is not expired and the
registration limit is not reached.
    - add_tags: This function adds theater id as tags to the instance.
    - update_table: This function updates the instance information in
dynamodb table.
    - get_activation_code_id: This function gets the activation code and
id from SSM parameter store.
    - lambda_handler: This function is the entry point for the lambda
function.

"""

import boto3
import json
import os
from datetime import datetime, timedelta
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools import Logger

# initialize powertools
tracer = Tracer()
metrics = Metrics()
logger = Logger()

# initialize boto3 client and resource
ssm_client = boto3.client("ssm")
dynamodb = boto3.resource("dynamodb")

# initialize environment variables
region = os.environ.get('AWS_REGION')
role = os.environ.get('SSM_ROLE')
dynamodb_table = os.environ.get('DYNAMODB_TABLE_ACTIVATIONS')
project = os.environ.get('PROJECT_NAME')
parameter = os.environ.get('SSM_PARAMETER')
install_dependencies = os.environ.get('INSTALL_DEPENDENCIES')


@tracer.capture_method(capture_response=False)
```

```
def install_deps(instance_id):
    """
    This function installs dependencies on the instance.

    Parameters:
        instance_id (str): Instance id of the instance

    Returns:
        response (dict): Response from SSM command

    Example:
        >>> response = install_deps(instance_id)
    """
    try:
        response = ssm_client.send_command(
            InstanceIds=[instance_id],
            DocumentName=install_dependencies,
            DocumentVersion="$LATEST",
            TimeoutSeconds=600,
            Comment="Installing dependencies"
        )
    except Exception as e:
        logger.error(e)
        raise e


@tracer.capture_method(capture_response=False)
def create_activation(registration_limit):
    """
    This function creates activation code and id for SSM agent and
stores it in SSM parameter store.

    Parameters:
        registration_limit (int): Registration limit for the activation
code

    Returns:
        activation_code (str): Activation code for SSM agent
        activation_id (str): Activation id for SSM agent

    Example:
        >>> activation_code, activation_id = create_activation(10)
    """

    try:
        response = ssm_client.create_activation(
            Description="ExpeDat SSM Agent Activation",
            DefaultInstanceName="TheaterBoxDCinema",
            IamRole=role,
            RegistrationLimit=registration_limit,
        )
    except Exception as e:
        logger.error(e)
        raise e

    return response["ActivationCode"], response["ActivationId"]
```

```
@tracer.capture_method(capture_response=False)
def update_ssm(activation_code, activation_id, count,
registration_limit, expiration):
    """
    This function updates the activation code and id in SSM parameter
store if the activation code is not expired and the registration limit
is not reached.

    Parameters:
        activation_code (str): Activation code for SSM agent
        activation_id (str): Activation id for SSM agent
        count (int): Active count for the activation code
        registration_limit (int): Registration limit for the activation
code
        expiration (str): Expiration date for the activation code

    Returns:
        response (dict): Response from SSM parameter store

    Example:
        >>> response = update_ssm(activation_code, activation_id, count,
registration_limit, expiration)
    """
    try:
        response = ssm_client.put_parameter(
            Name=parameter,
            Value=json.dumps({
                "ActivationId": activation_id,
                "ActivationCode": activation_code,
                "Expiration": expiration,
                "ActiveCount": count,
                "RegistrationLimit": registration_limit
            }),
            Type="String",
            Overwrite=True
        )
    except Exception as e:
        logger.error(e)
        raise e
    return response


@tracer.capture_method(capture_response=False)
def add_tags(instance_id, theater_id, theater_name, theater_email):
    """
    This function adds theater id as tags to the instance.

    Parameters:
        instance_id (str): Instance id of the instance
        theater_id (str): Theater id of the instance

    Returns:
        response (dict): Response from SSM parameter store

    Example:
        >>> response = add_tags(instance_id, theater_id)
```

```
    """
    try:
        response = ssm_client.add_tags_to_resource(
            ResourceType="ManagedInstance",
            ResourceId=instance_id,
            Tags=[
                {
                    "Key": "TheaterId",
                    "Value": theater_id
                },
                {
                    "Key": "TheaterName",
                    "Value": theater_name
                },
                {
                    "Key": "UpdatedOn",
                    "Value": datetime.now().strftime("%Y-%m-%d %H:%M:
%S")

                },
                {
                    "Key": "TheaterEmail",
                    "Value": theater_email
                },
                {
                    "Key": "Project",
                    "Value": project
                }
            ]
        )
    except Exception as e:
        logger.error(e)
        raise e
    return response


@tracer.capture_method(capture_response=False)
def update_table(theater_id, theater_name, theater_email, instance_id):
    """
    This function adds theater id's tags to the theater table.

    Parameters:
        instance_id (str): Instance id of the instance
        theater_id (str): Theater id of the instance

    Returns:
        response (dict): Response from SSM parameter store

    Example:
        >>> response = add_tags(instance_id, theater_id, theater_name,
theater_email)
    """
    try:
        table = dynamodb.Table(os.environ.get('DYNAMODB_TABLE_THEATER'))

        response = table.put_item(
            Item={
```

```
                'TheaterId': theater_id,
                'TheaterName': theater_name,
                'TheaterEmail': theater_email,
                'InstanceId': instance_id,
                'UpdatedOn': datetime.now().strftime("%Y-%m-%d %H:%M:
%S")

            }
        )
    except Exception as e:
        logger.error(e)
        raise e
    return response


@tracer.capture_method(capture_response=False)
def get_activation_code_id():
    """
    This function gets activation code and id from SSM parameter store
if the activation code is not expired and the registration limit is not
reached. If the activation code is expired or the registration limit is
reached, it creates a new activation code and id and stores it in SSM
parameter store.

    Returns:
        activation_code (str): Activation code for SSM agent
        activation_id (str): Activation id for SSM agent

    Example:
        >>> activation_code, activation_id = get_activation_code_id()
    """
    try:
        response = ssm_client.get_parameter(
            Name=parameter,
            WithDecryption=False
        )
        item = json.loads(response["Parameter"]["Value"])
        count = item.get("ActiveCount", 0) + 1
        registration_limit = item["RegistrationLimit"]
        expiration_date = datetime.strptime(
            item["Expiration"], '%Y-%m-%d').date()
        if expiration_date > datetime.now().date():
            if count < registration_limit:
                ssm_client.put_parameter(
                    Name=parameter,
                    Value=json.dumps({
                        "ActivationId": item["ActivationId"],
                        "ActivationCode": item["ActivationCode"],
                        "Expiration": item["Expiration"],
                        "ActiveCount": count,
                        "RegistrationLimit": registration_limit
                    }),
                    Type="String",
                    Overwrite=True
                )
                logger.info("Updated ActiveCount in ssm parameter")
                return item["ActivationCode"], item["ActivationId"]
```

```
            else:
                activation_code, activation_id = create_activation(
                    registration_limit)
                expiration = (datetime.now() +
                              timedelta(days=28)).strftime("%Y-%m-%d")
                update_ssm(activation_code, activation_id,
                           0, registration_limit, expiration)
                logger.info("Created new activation code and id")
                return activation_code, activation_id
        else:
            activation_code, activation_id = create_activation(
                registration_limit)
            expiration = (datetime.now() + timedelta(days=28)
                          ).strftime("%Y-%m-%d")
            update_ssm(activation_code, activation_id,
                       0, registration_limit, expiration)
            logger.info("Created new activation code and id")
            return activation_code, activation_id
    except Exception as e:
        logger.error(e)
        raise e


@logger.inject_lambda_context(log_event=True)
@tracer.capture_lambda_handler
def lambda_handler(event: dict, context: LambdaContext):
    """
    This function is the entry point for the lambda function. It checks
the api path and calls the respective function.

    Parameters:
        event (dict): Event data passed to the lambda function
        context (LambdaContext): Lambda context object

    Returns:
        response (dict): Response from the function

    """

    # check api path to see if the request is for activation code and id
or to add theater id as tags to the instance
    event_path = event["path"]
    if event_path == "/addtags":
        # add theater id as tags to the instance
        body = json.loads(event["body"])
        instance_id = body["machine_id"]
        theater_id = body["theater_id"]
        theater_name = body["theater_name"]
        theater_email = body["theater_email"]
        try:
            tag_response = add_tags(instance_id, theater_id,
                                    theater_name, theater_email)
            logger.info("Added tags to the instance")

            # add theater id, name and instance id to dynamodb table
            table_response = update_table(
                theater_id, theater_name, theater_email, instance_id)
```

```
            # run install dependencies ssm run document on the instance
            ssm_response = install_deps(instance_id)
            logger.info("Ran install dependencies ssm run document")

            return {
                "statusCode": 200,
                "body": json.dumps(
                    {
                        "Message": "TheaterId tag added successfully to
the instance",
                        "TheaterId": theater_id,
                        "InstanceId": instance_id,
                    }
                ),
            }
        except Exception as e:
            logger.error(e)
            return {
                "statusCode": 500,
                "body": json.dumps(
                    {
                        "Message": "Error adding tags to the instance"
                    }
                )
            }

    # get activation code and id
    elif event_path == "/ssmactivation":
        activation_code, activation_id = get_activation_code_id()
        return {
            "statusCode": 200,
            "body": json.dumps(
                {
                    "ActivationId": activation_id,
                    "ActivationCode": activation_code,
                    "Region": region,
                }
            )
        }
    else:
        return {
            "statusCode": 404,
            "body": json.dumps(
                {
                    "Message": "Invalid path"
                }
            )
        }
```

SSMInstallScriptGenerator

```
"""
This function is used to generate the SSM install script for the
customer
"""

from jinja2 import Environment, FileSystemLoader
import boto3
import os
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all
patch_all()


@xray_recorder.capture('lambda_handler')
def lambda_handler(event, context):
    """
    This function is used to generate the SSM install script for the
customer
    """
    # get the template file
    env = Environment(loader=FileSystemLoader('.'))
    template = env.get_template('sample.py')
    bucket = os.environ.get('BUCKET')
    # Render the template with the environment variable
    OT_URL = os.environ.get('ONBOARD_TENANT_URL')
    SA_URL = os.environ.get('SAVE_ACTIVATION_URL')
    rendered_template = template.render(
        ONBOARD_TENANT_URL=OT_URL, SAVE_ACTIVATION_URL=SA_URL)

    # send response to api for file download
    with open("/tmp/install_ssmAgent.py", "w") as f:
        f.write(rendered_template)

    with open("/tmp/install_ssmAgent.py", "rb") as f:
        file_data = f.read()
    response = {
        "statusCode": 200,
        "headers": {
            "Content-Type": "application/octet-stream",
            "Content-Disposition": "attachment;
filename=install_ssmAgent.py"
        },
        "body": file_data
    }

    os.remove("/tmp/install_ssmAgent.py")
    return response
```

SSMInstallScriptGenerator

```
# ask input from user for tenant name
# use the input and call onboard tenant API to get activationID and
activationCode as response
# download amazon-ssm-agent.deb and install it using dpkg -i amazon-ssm-
agent.deb
# stop the service amazon-ssm-agent stop
# use the activationID and activationCode to register the tenant by
running sudo -E amazon-ssm-agent -register -code $code -id $id -region
$region
# start the service amazon-ssm-agent start
# create cloudwatch alarms using api call
# collect hostname, ip and use storage.json from local directory to
store the data and call api to store the data
# exit the script

import json
import os
import sys
import time
import subprocess
import logging
import argparse
from subprocess import Popen, PIPE


def prRed(prt):
    print("\033[91m {}\033[00m".format(prt))


def prGreen(prt):
    print("\033[92m {}\033[00m".format(prt))


def prYellowC(prt):
    print("\033[93m {}\033[00m".format(prt))


def prYellow(prt):
    print("\33[5m {}\033[00m".format(prt))


def get_args():
    parser = argparse.ArgumentParser(description="Process tenant name")
    parser.add_argument("-t", "--tenant", help="tenant name",
required=True)
    args = parser.parse_args()
    return args


def get_activation_details(theater_id, os_name, theater_name):
    prYellow("Getting activation details")
    import requests
```

```
    url = "{{ ONBOARD_TENANT_URL }}"
    # url = "https://pq5ymlotdc.execute-api.ap-south-1.amazonaws.com/
Prod/ssmactivation"
    payload = {"theater_id": theater_id,
               "os": os_name, "theater_name": theater_name}
    response = requests.post(url, json=payload)
    if response.status_code == 200:
        prGreen("Successfully got activation details")
        print(response.json())
        return response.json()
    else:
        prRed("Error: {}".format(response.json()))
        sys.exit(1)


def install_ssm_agent(activation_details, theater_id, theater_name,
theater_email):
    prYellow("Starting installation of ssm agent")
    import requests

    activation_id = activation_details["ActivationId"]
    activation_code = activation_details["ActivationCode"]
    region = activation_details["Region"]
    # check if os is linux or windows and download the ssm agent
accordingly
    if os.name == "posix":
        prYellow("Downloading ssm agent for linux")
        subprocess.call(
            [
                "wget",
                "https://s3.amazonaws.com/ec2-downloads-windows/
SSMAgent/latest/debian_amd64/amazon-ssm-agent.deb",
            ]
        )
        subprocess.call(["sudo", "dpkg", "-i", "amazon-ssm-agent.deb"])
        prGreen("Successfully downloaded ssm agent")
        # install the package
        prYellow("Installing ssm agent")

        subprocess.call(["sudo", "dpkg", "-i", "amazon-ssm-agent.deb"])
        prGreen("Successfully installed ssm agent")
        stop_ssm_agent()
        # get hostname
        hostname = subprocess.check_output(["hostname"]).decode("utf-8")

        # use the activationID and activationCode to register the tenant
by running sudo -E amazon-ssm-agent -register -code $code -id $id -
region $region
        command = "sudo -E amazon-ssm-agent -register -code {} -id {} -
region {}".format(
            activation_code, activation_id, region
        )
        os.system(command)
        instanceID = ""
        path = '/var/lib/amazon/ssm/registration'
        if os.path.exists(path):
```

```
            with open(path, 'r') as f:
                instance_data = json.load(f)
                instance_id = instance_data['ManagedInstanceID']
            instanceID = instance_id
            test = save_registration_details(
                instanceID, theater_id, theater_name, theater_email,
hostname)
        else:
            prRed("Error: Registration file not found")
            sys.exit(1)

        prGreen("Successfully registered Theater")
        # clean_up()
    elif os.name == "nt":

        tempPath = os.getenv("TEMP")

        subprocess.call(
            [
                "powershell",
                "Invoke-WebRequest",
                "https://amazon-ssm-ap-south-1.s3.ap-
south-1.amazonaws.com/latest/windows_amd64/AmazonSSMAgentSetup.exe",
                "-OutFile",
                ".\AmazonSSMAgentSetup.exe",
            ]
        )
        # Start-Process .\AmazonSSMAgentSetup.exe -ArgumentList @("/q",
"/log", "install.log", "CODE=$code", "ID=$id", "REGION=$region") -Wait

        subprocess.call(
            [
                "powershell",
                "Start-Process",
                ".\AmazonSSMAgentSetup.exe",
                "-ArgumentList",
                '@("/q", "/log", "install.log", "CODE={}", "ID={}",
"REGION={}")'.format(
                    activation_code, activation_id, region
                ),
                "-Wait",
            ]
        )
        hostname = subprocess.check_output(["hostname"]).decode("utf-8")
        with open(os.getenv("PROGRAMDATA") + "\\Amazon\\SSM\
\InstanceData\\registration", "r") as f:
            data = f.read()
            data = json.loads(data)
            instance_id = data["ManagedInstanceID"]
            region = data["Region"]
            print(instance_id)
        test = save_registration_details(
            instance_id, theater_id, theater_name, theater_email,
hostname)
        # Get-Content ($env:ProgramData +
"\Amazon\SSM\InstanceData\registration")
        # subprocess.call(
```

```
        #     [
        #         "powershell",
        #         "Get-Content",
        #         os.getenv("ProgramData") +
        #         "\Amazon\SSM\InstanceData\registration",
        #     ]
        # )
        # # Get-Service -Name "AmazonSSMAgent"
        # subprocess.call(["powershell", "Get-Service",
        #                  "-Name", "AmazonSSMAgent"])
        # clean_up()
    else:
        prRed("Error: OS not supported")
        sys.exit(1)
    # download amazon-ssm-agent.deb from https://s3.amazonaws.com/ec2-
downloads-windows/SSMAgent/latest/debian_amd64/amazon-ssm-agent.deb
    # download from url
    # url = "https://s3.amazonaws.com/ec2-downloads-windows/SSMAgent/
latest/debian_amd64/amazon-ssm-agent.deb"
    # r = requests.get(url, allow_redirects=True)
    # open('amazon-ssm-agent.deb', 'wb').write(r.content)


# check up function to delete amazon-ssm-agent.deb , get-pip.py and
storage.json. Uninstall the package requests and delete the file


def clean_up():
    prYellow("Cleaning up")
    if os.name == "posix":
        # remove amazon-ssm-agent.deb, get-pip.py and storage.json if
present
        if os.path.isfile("amazon-ssm-agent.deb"):
            os.remove("amazon-ssm-agent.deb")
            prGreen("Successfully removed amazon-ssm-agent.deb")
        if os.path.isfile("get-pip.py"):
            os.remove("get-pip.py")
            prGreen("Successfully removed get-pip.py")
        if os.path.isfile("storage.json"):
            os.remove("storage.json")
            prGreen("Successfully removed storage.json")
        # uninstall requests
        subprocess.call(["sudo", "pip", "uninstall", "-y", "requests"])
        print(
            "\033[48;5;236m\033[38;5;231mThe script seems to have
completed successfully, please check the status of SSM agent by running
\033[38;5;208m'systemctl status amazon-ssm-agent'\033[0;0m"
        )
        dir = os.getcwd()
        os.remove(dir + "/%s" % sys.argv[0])
        sys.exit(1)
    elif os.name == "nt":
        # os.getenv("TEMP") + '\SSMAgent_latest.exe'

        # remove amazon-ssm-agent.deb, get-pip.py and storage.json if
present
        if os.path.isfile(os.getenv("TEMP") + "\SSMAgent_latest.exe"):
```

```
            os.remove(os.getenv("TEMP") + "\SSMAgent_latest.exe")
            prGreen("Successfully removed amazon-ssm-agent.deb")
        if os.path.isfile(os.getenv("TEMP") + "\get-pip.py"):
            os.remove(os.getenv("TEMP") + "\get-pip.py")
            prGreen("Successfully removed get-pip.py")
        if os.path.isfile(os.getenv("TEMP") + "\storage.json"):

            os.remove(os.getenv("TEMP") + "\storage.json")
            prGreen("Successfully removed storage.json")
        # uninstall requests
        subprocess.call(["powershell", "pip", "uninstall", "-y",
"requests"])
        print(
            "\033[48;5;236m\033[38;5;231mThe script seems to have
completed successfully, please check the status of SSM agent by running
\033[38;5;208m'systemctl status amazon-ssm-agent'\033[0;0m"
        )
        dir = os.getcwd()
        os.remove(dir + "/%s" % sys.argv[0])
        sys.exit(1)
    else:
        prRed("Error: OS not supported")
        sys.exit(1)


def stop_ssm_agent():
    prYellow("Stopping ssm agent")
    subprocess.call(["sudo", "amazon-ssm-agent", "stop"])


def start_ssm_agent():
    prYellow("Starting ssm agent")
    subprocess.call(["sudo", "amazon-ssm-agent", "start"])


def save_registration_details(machine_id, theater_id, theater_name,
theater_email, hostname):
    prYellow("Saving activation details")
    import requests

    url = "{{ SAVE_ACTIVATION_URL }}"
    # url = "https://pq5ymlotdc.execute-api.ap-south-1.amazonaws.com/
Prod/ssmactivation"
    payload = {"machine_id": machine_id,
               "theater_id": theater_id, "theater_name": theater_name,
"theater_email": theater_email, "hostname": hostname}
    response = requests.post(url, json=payload)
    if response.status_code == 200:
        prGreen("Successfully saved activation details")
        print(response.json())
        return response.json()
    else:
        prRed("Error: {}".format(response.json()))
        sys.exit(1)


def run():
```

```
    try:

        if os.name == "posix":
            os_name = "LINUX/OnPrem"
        elif os.name == "nt":
            os_name = "WINDOWS/OnPrem"

        # ask for theater id and tenant name from user
        # theater_id = input("Enter theater id: ")
        theater_id = str(input("Enter theater id: "))
        if theater_id.isdigit():
            raise ValueError("Theater id should not be a number")

        theater_name = str(input("Enter theater name: "))
        theater_email = str(input("Enter theater email: "))

        # theater_id = "theater-1"

        # tenant_name = "Services-Tenant"
        activation_details = get_activation_details(
            theater_id, os_name, theater_name)
        # check os path /lib/systemd/system/amazon-ssm-agent.service
        if os.name == "posix":
            if os.path.exists("/lib/systemd/system/amazon-ssm-
agent.service"):
                prGreen("SSM agent is already installed")
                # create_cloudwatch_alarms("storage.json")
                reRegister(activation_details, theater_id, theater_name)
                # clean_up()
            else:
                prRed("SSM agent is not installed")
                install_ssm_agent(activation_details, theater_id,
                                  theater_name, theater_email)
                sys.exit(1)
        elif os.name == "nt":
            # check for path C:\ProgramData\Amazon\SSM\InstanceData
            hostanme = subprocess.check_output(
                ["powershell", "hostname"]).decode("utf-8").strip()

            if os.path.exists(os.getenv("PROGRAMDATA") +
"\Amazon\SSM\InstanceData"):
                # read file registration and parse the json to get
instanceID from
{"ManagedInstanceID":"mi-065997545cc7a09b3","Region":"ap-south-1"}

                with open(os.getenv("PROGRAMDATA") + "\\Amazon\\SSM\
\InstanceData\\registration", "r") as f:
                    data = f.read()
                    data = json.loads(data)
                    instance_id = data["ManagedInstanceID"]
                    region = data["Region"]
                    print(instance_id)
                test = save_registration_details(
                    instance_id, theater_id, theater_name,
theater_email, hostanme)
                prGreen(
                    "SSM agent is already installed and the machine ID
```

```
is {}".format(instance_id))
                # create_cloudwatch_alarms("storage.json")
                # reRegister(activation_details, theater_id,
theater_name)
                # clean_up()
            else:
                prRed("SSM agent is not installed")
                install_ssm_agent(activation_details, theater_id,
                                  theater_name, theater_email)
                sys.exit(1)
        else:
            prRed("Error: OS not supported")
            sys.exit(1)

    except Exception as e:
        prRed("Error: {}".format(e))
        sys.exit(1)


def reRegister(activation, theater_id, theater_name):
    activation_id = activation["ActivationId"]
    activation_code = activation["ActivationCode"]
    region = "ap-south-1"

    service = "amazon-ssm-agent"
    if os.name == "posix":
        # use os.system to run the command
        command = "echo yes | sudo amazon-ssm-agent -register -code {} -
id {} -region ap-south-1".format(
            activation_code, activation_id
        )
        os.system(command)
        # restart the service
        command = "sudo systemctl restart {}".format(service)
        os.system(command)

        # check if the service is active
        isActive = subprocess.check_output(
            ["systemctl", "is-active", "amazon-ssm-agent"]
        )
        isActive = isActive.decode("utf-8")
        isActive = isActive.strip()
        hostname = subprocess.check_output(
            ["hostname"]).decode("utf-8").strip()

        if isActive == "active":

            instanceID = ""
            path = '/var/lib/amazon/ssm/registration'
            if os.path.exists(path):
                with open(path, 'r') as f:
                    instance_data = json.load(f)
                    instance_id = instance_data['ManagedInstanceID']
                instanceID = instance_id
                test = save_registration_details(
                    instanceID, theater_id, theater_name, theater_email,
hostname)
```

```
            prGreen("SSM agent is active")

        else:
            prRed("SSM agent is not installed")
            sys.exit(1)
    elif os.name == "nt":
        # use subprocess.call to run the command 'yes' | & 'C:\Program
Files\Amazon\SSM\amazon-ssm-agent.exe' -register -code activation-code -
id activation-id -region region; Restart-Service AmazonSSMAgent
        subprocess.call(
            [
                "yes",
                "|",
                "C:\\Program Files\\Amazon\\SSM\\amazon-ssm-agent.exe",
                "-register",
                "-code",
                activation_code,
                "-id",
                activation_id,
                "-region",
                region,
            ]
        )

        sys.exit(1)


def main():
    # check if package requests is installed if not install it and then
call all the functions

    prYellow("Checking requirements")
    try:
        import requests

        prGreen("All required modules available")
        prYellowC(
            "Do you want to install ssm agent? \033[38;5;208m Type '1'
for YES / '2' for NO :\033[0;0m"
        )
        choice = input()
        if choice == "1":
            run()
        elif choice == 2:
            prYellowC(
                "Do you want to create cloudwatch alarms? \033[38;5;208m
Type '1' for YES / '2' for NO :\033[0;0m"
            )
            choice = input()
            if choice == "1":
                tenant_list = ["Services-Tenant", "Star-Tenant"]
                # create menu to select tenant
                print(
                    "\033[48;5;236m\033[38;5;231mSelect the tenant to
register with SSM agent\033[0;0m"
                )
                for i, tenant in enumerate(tenant_list):
```

```
                    print(
                        "\033[48;5;236m\033[38;5;231m{}. {}
\033[0;0m".format(
                            i + 1, tenant
                        )
                    )
                tenant_choice = int(
                    input(
                        "\033[48;5;236m\033[38;5;231mEnter your choice:
\033[0;0m")
                )
                if tenant_choice > len(tenant_list):
                    prRed("Invalid choice")
                    sys.exit(1)
                tenant_name = tenant_list[tenant_choice - 1]
                # clean_up()

            elif choice == 2:
                sys.exit(1)
            else:
                prRed("Wrong input")
                sys.exit(1)
        else:
            prRed("Wrong input")
            sys.exit(1)
    except ImportError:
        prRed("Requests package is not installed. Installing it now")
        # install pip using curl https://bootstrap.pypa.io/pip/2.7/get-
pip.py --output get-pip.py
        # run python get-pip.py
        url = "https://bootstrap.pypa.io/pip/2.7/get-pip.py"
        subprocess.call(["curl", url, "--output", "get-pip.py"])
        subprocess.call(["python", "get-pip.py"])

        subprocess.call(["pip", "install", "requests"])
        prGreen("Successfully installed requests")
        import requests

        # ask user input of 1 or 2 to install ssm agent or to run
create_cloudwatch_alarms function
        prYellowC(
            "Do you want to install ssm agent? \033[38;5;208m Type '1'
for YES / '2' for NO :\033[0;0m"
        )
        choice = input()
        if choice == "1":
            run()
        elif choice == 2:
            prYellowC(
                "Do you want to create cloudwatch alarms? \033[38;5;208m
Type '1' for YES / '2' for NO :\033[0;0m"
            )
            choice = input()
            if choice == "1":
                tenant_list = ["Services-Tenant", "Star-Tenant", "New-
Tenant"]
                # create menu to select tenant
```

```
                print(
                    "\033[48;5;236m\033[38;5;231mSelect the tenant to
register with SSM agent\033[0;0m"
                )
                for i, tenant in enumerate(tenant_list):
                    print(
                        "\033[48;5;236m\033[38;5;231m{}. {}
\033[0;0m".format(
                            i + 1, tenant
                        )
                    )
                tenant_choice = int(
                    input(
                        "\033[48;5;236m\033[38;5;231mEnter your choice:
\033[0;0m")
                )
                if tenant_choice > len(tenant_list):
                    prRed("Invalid choice")
                    sys.exit(1)
                tenant_name = tenant_list[tenant_choice - 1]
                # clean_up()
            elif choice == 2:
                sys.exit(1)
            else:
                prRed("Wrong input")
                sys.exit(1)
        else:
            prRed("Wrong input")
            sys.exit(1)


if __name__ == "__main__":
    main()
    sys.exit(1)


# vim: tabstop=8 expandtab shiftwidth=4 softtabstop=4
# vi: set ft=python:
# EOF
```

SSMNotification

```
"""
This file contains the error codes and error messages for ExpeDat .
For more information, see the ExpeDat documentation at https://
www.dataexpedition.com/expedat/Docs/movedat/exit-codes.html .

"""
error_dict = {
    0: "Success: all requested operations completed successfully",
    1: "Could not determine the nature of the error",
    4: "An unsupported feature was requested",
    5: "Invalid address",
    8: "Object too large to be delivered",
    9: "Object unavailable",
    10: "Bad Credentials (username or password)",
    11: "Object is busy or locked - try again later",
    13: "Operation timed out",
    14: "A requested condition was not met",
    17: "Unsupported application version",
    18: "Invalid Argument",
    20: "Transaction lost or unrecognized",
    21: "Encryption required",
    22: "Requested encryption is not supported",
    23: "Requested key is not valid or not supported",
    24: "Request denied by configuration or user input",
    25: "Invalid pathname",
    26: "A server name is invalid or no server could be reached",
    27: "Insufficient privileges for requested action",
    28: "Requested feature not supported",
    29: "Operating system error",
    30: "Server capacity has been exceeded",
    31: "Exceeded resource limit based on credentials",
    32: "Session aborted",
    33: "Out of memory",
    34: "Directory already exists or requested state already set",
    36: "A partial upload was detected and resumption was not selected",
    37: "A partial upload was detected, but the file or meta data
appears to be corrupted",
    38: "A partial upload was detected, but the source and destination
files appear to be different",
    43: "A feature was requested which is not supported by the software
license",
    66: "Error in network system call",
    69: "Network buffer overflow",
    71: "Request expired because network connectivity was lost",
    72: "Address is not valid",
    74: "Port is not valid (out of range)",
    76: "ICMP Network is down",
    77: "ICMP Host is down",
    78: "ICMP No application on given port",
    80: "ICMP Network unknown",
    81: "ICMP Host unknown",
    82: "ICMP Net/Host/Filter Prohibited",
```

```
    248: "Problem with command line or configuration file",
    249: "Error accessing a local file",
    250: "An error occurred with the DEI toolkit",
    251: "An error occurred with the SEQ module",
    252: "An error occurred with the DOC module",
    253: "An error occurred with the MTP module",
    254: "License/registration problem",
    255: "Multiple actions were requested and some failed"
}


def get_error_message(error_message):
    # "failed to run commands: exit status 255"
    # extract the exit code from the error message string
    exit_code = None
    try:
        if error_message and isinstance(error_message, str):
            exit_code_start_index = error_message.find(
                'exit status ') + len('exit status ')
            if exit_code_start_index >= len('exit status '):
                exit_code_str =
error_message[exit_code_start_index:].split()[
                    0]
                if exit_code_str.isdigit():
                    exit_code = int(exit_code_str)
    except Exception as e:
        print(f"Exception in get_error_message: {e}")
    if exit_code is not None:
        return error_dict.get(exit_code, "Could not determine the nature
of the error")
    else:
        return error_message
```

SSMNotification

```
"""
This function is triggered by SNS topic when SSM document execution is
completed, failed or timed out.
This function will fetch the instance id from the SNS message and fetch
the theater details from dynamodb table using instance id.
This function will fetch the SSM document execution details from SSM
using command id.
This function will send an email to the theater email id with the SSM
document execution details.
This function will store the SSM document execution details in dynamodb
table.

Functions:
    - lambda_handler: The main function of the Lambda function. This
function is triggered by an SNS topic. The function uses the AWS Systems
Manager (SSM) API to query the status of the SSM Agent on the EC2
instances.
    - verify_email_identity: Verifies the email identity.
    - get_instance_details: Gets the theater details from dynamodb table
using instance id.
    - get_command_details: Gets the SSM document execution details from
SSM using command id.
    - send_notification: Sends an email to the theater email id with the
SSM document execution details.
    - store_status_in_dynamodb: Stores the SSM document execution
details in dynamodb table.
    - clean_data: Cleans the SSM document execution details.


"""
import json
import os
import re
import csv
import io
import boto3
import botocore
from aws_lambda_powertools import Logger
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from jinja2 import Environment, FileSystemLoader
import datetime
import time
import pytz
from exit_codes import get_error_message
import base64
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email.mime.image import MIMEImage

# initialize tracer, metrics and loggers
tracer = Tracer()
```

```
metrics = Metrics()
logger = Logger()

# initialize boto3 client and resource
dynamodb = boto3.resource("dynamodb")
dynamodb_table = os.environ.get('DYNAMODB_TABLE')
# dynamodb_table_index = os.environ.get('DYNAMODB_TABLE_INDEX')
dynamodb_table_index = "InstanceIdIndex"
emailfrom = os.environ.get('EMAIL_FROM')
table = dynamodb.Table(dynamodb_table)
ssm_client = boto3.client("ssm")
ses_client = boto3.client("ses")
status_table = os.environ.get('STATUS_TABLE')
s3_client = boto3.client("s3")
date = datetime.datetime.now().strftime("%Y-%m-%d")
s3_bucket = os.environ.get('S3_BUCKET')
# Read the first image file
with open('template/images/logo.png', 'rb') as f1:
    img_data1 = f1.read()

# Read the second image file
with open('template/images/header.png', 'rb') as f2:
    img_data2 = f2.read()

# initialize environment variables
role = os.environ.get('Role')
documeName = os.environ.get('SSM_DOCUMENT_NAME')

env = Environment(loader=FileSystemLoader('.'))
ist_tz = pytz.timezone('Asia/Kolkata')

# function to store status of the SSM document execution in dynamodb
table


@tracer.capture_method(capture_response=False)
def store_status_in_dynamodb(command_details, theater_id, message):
    """
    Store the status of the SSM document execution in dynamodb table.

    Parameters
    ----------
    command_details : dict
        SSM document execution details.
    theater_id : str
        Theater id.
    message : dict
        SNS message.

    Returns
    -------
    bool
        True if status is stored in dynamodb table else False.
    str
        Error message if status is not stored in dynamodb table.

    """
```

```
    dtable = dynamodb.Table(status_table)
    folder_name, info, start_time, end_time, current_time, status,
command_id, output, error, source_path, dest_path, source_hash,
dest_hash = clean_data(
        command_details, message)
    created_on = int(time.time())
    item = {
        "TheaterId": theater_id,
        "JobId": command_id,
        "Status": status,
        "StartTime": start_time,
        "CreatedOn": created_on,
        "FolderName": folder_name,
        "SourcePath": source_path,
        "DestinationPath": dest_path,
    }

    if status == "Success":
        item["EndTime"] = end_time
        item["Info"] = info
        item["Output"] = output
        item["SourceHash"] = source_hash
        item["DestinationHash"] = dest_hash
    elif status == "Failed":
        item["EndTime"] = end_time
        item["Error"] = error
    elif status == "TimedOut":
        item["EndTime"] = current_time
        item["Error"] = error
    else:
        item["TTL"] = created_on + 86400

    try:
        dtable.put_item(Item=item)
    except botocore.exceptions.ClientError as e:
        logger.error(e.response)
        return None, json.dumps(e.response)
    return True, None


# def extract_info(text):
#     last_line = text.strip().split('\n')[-1]
#     if last_line[0] == 'F':
#         info = last_line.split('\t')[-2]
#         return info
#     return None
def format_time(time_sec):
    if time_sec < 60:
        return '{:.1f} seconds'.format(time_sec)
    elif time_sec < 3600:
        min = int(time_sec // 60)
        sec = int(time_sec % 60)
        return '{} minutes {} seconds'.format(min, sec)
    else:
        hour = int(time_sec // 3600)
        min = int((time_sec % 3600) // 60)
```

```
        sec = int((time_sec % 3600) % 60)
        return '{} hours {} minutes {} seconds'.format(hour, min, sec)


def extract_info(log):
    try:
        # Total number of files: 40
        # Total size of files (in GB): 40.80000000000002
        # Total time taken (in seconds): 20568.0
        # Best speed (in Mbps): 46.1
        # Average speed (in Mbps): 23.575249999999997
        # Destination Path: /tmp/DCinema/downloads/TestPackaged40GB
        # Source Path: /ldc1/dc-storage/DCinemaTest/TestPackage40GB

        pattern_size = r'Total size of files \(in ([A-Za-z]+)\):
([0-9.]+)'
        pattern_num_files = r'Total number of files: (.+)'
        pattern_time_taken = r'Total time taken \(in seconds\): (.+)'
        pattern_speed = r'Best speed \(in Mbps\): (.+)'
        pattern_avg_speed = r'Average speed \(in Mbps\): (.+)'
        pattern_dest_path = r'Destination Path: (.+)'
        pattern_source_path = r'Source Path: (.+)'
        pattern_source_hash = r'Server Key: (.+)'
        pattern_dest_hash = r'Client Key: (.+)'

        size_match = re.search(pattern_size, log)
        if size_match is None:
            raise ValueError('Invalid log format: cannot extract file
size')
        num_files_match = re.search(pattern_num_files, log)
        if num_files_match is None:
            raise ValueError(
                'Invalid log format: cannot extract number of files')

        time_taken_match = re.search(pattern_time_taken, log)
        if time_taken_match is None:
            raise ValueError('Invalid log format: cannot extract time
taken')
        speed_match = re.search(pattern_speed, log)
        if speed_match is None:
            raise ValueError('Invalid log format: cannot extract best
speed')
        avg_speed_match = re.search(pattern_avg_speed, log)
        if avg_speed_match is None:
            raise ValueError(
                'Invalid log format: cannot extract average speed')
        source_path_match = re.search(pattern_source_path, log)
        if source_path_match is None:
            raise ValueError(
                'Invalid log format: cannot extract source path')
        dest_path_match = re.search(pattern_dest_path, log)
        if dest_path_match is None:
            raise ValueError(
                'Invalid log format: cannot extract destination path')
        source_hash_match = re.search(pattern_source_hash, log)
        if source_hash_match is None:
            raise ValueError(
```

```
                'Invalid log format: cannot extract source hash')
        dest_hash_match = re.search(pattern_dest_hash, log)
        if dest_hash_match is None:
            raise ValueError(
                'Invalid log format: cannot extract destination hash')

        size_unit = size_match.group(1)
        size_val = float(size_match.group(2))
        if size_unit.lower() == 'kb':
            size_val /= 1024**2
        elif size_unit.lower() == 'mb':
            size_val /= 1024
        elif size_unit.lower() == 'gb':
            pass
        else:
            return None

        num_files = int(num_files_match.group(1))
        time_taken = float(time_taken_match.group(1))
        speed = float(speed_match.group(1))
        avg_speed = float(avg_speed_match.group(1))
        source_path = source_path_match.group(1)
        dest_path = dest_path_match.group(1)
        source_hash = source_hash_match.group(1)
        dest_hash = dest_hash_match.group(1)

        return size_val, num_files, time_taken, speed, avg_speed,
source_path, dest_path, source_hash, dest_hash
    except Exception as e:
        logger.error(e)
        return None


def clean_data(command_details, message):
    """
    Clean the SSM document execution details.

    Parameters
    ----------
    command_details : dict
        SSM document execution details.
    message : dict
        SNS message.

    Returns
    -------
    str
        Folder name.
    str
        Info.
    str
        Start time.
    str
        End time.
    str
        Current time.
    str
```

```
        Status.
    str
        Command id.
    str
        Output.
    str
        Error.

    Examples
    --------
    >>> clean_data(command_details, message)
    ('folder_name', 'info', 'start_time', 'end_time', 'current_time',
'status', 'command_id', 'output', 'error')

    """
    folder_name = None
    info = None
    end_time = None
    source_path = None
    dest_path = None
    source_hash = None
    dest_hash = None
    output = command_details["StandardOutputContent"]
    comments = command_details["Comment"]
    error = command_details["StandardErrorContent"]

    status = message["status"]
    command_id = message["commandId"]
    start_time = message["requestedDateTime"]
    start_time = datetime.datetime.strptime(
        start_time, "%Y-%m-%dT%H:%M:%S.%fZ")
    start_time = start_time.replace(tzinfo=pytz.utc).astimezone(
        ist_tz).strftime("%Y-%m-%d %I:%M:%S %p %Z")
    # Get the current UTC time
    now_utc = datetime.datetime.utcnow()

    # Convert UTC time to Indian Standard Time

    now_ist = pytz.utc.localize(now_utc).astimezone(ist_tz)
    current_time = now_ist.strftime("%Y-%m-%d %I:%M:%S %p %Z")

    if command_details["ExecutionEndDateTime"] is not None and
command_details["ExecutionEndDateTime"] != "":
        end_time = command_details["ExecutionEndDateTime"]
        end_time = datetime.datetime.strptime(
            end_time, "%Y-%m-%dT%H:%M:%S.%fZ")
        end_time = end_time.replace(tzinfo=pytz.utc).astimezone(
            ist_tz).strftime("%Y-%m-%d %I:%M:%S %p %Z")

    if output != "" and output != " ":
        # print(output)
        data = extract_info(output)
        source_path = data[5]
        dest_path = data[6]
        source_hash = data[7]
        dest_hash = data[8]
        if data:
```

```
            info = "Total size of files: {:.2f} GB, Total number of
files: {}, Total time taken: {}, Best speed: {:.2f} Mbps, Average speed:
{:.2f} Mbps".format(
                data[0], data[1], format_time(data[2]), data[3],
data[4])

        else:
            info = "No information available"

    if comments != "" and comments != " ":
        match = re.search(r'Copy from (\S+) to', comments)
        if match:
            folder_name = match.group(1)

    return folder_name, info, start_time, end_time, current_time,
status, command_id, output, error, source_path, dest_path, source_hash,
dest_hash


# function to fetch instance id from dynamodb table using theater_id and
return instance id for each theater


@tracer.capture_method(capture_response=False)
def get_instance_details(instance_id, table):
    """
    Fetch instance id from dynamodb table using theater_id and return
instance id for each theater.

    Parameters
    ----------
    instance_id : str
        Instance id.
    table : str
        DynamoDB table name.

    Returns
    -------
    list
        List of instance ids.
    str
        Error message if instance id is not fetched from dynamodb table.

    """

    try:
        response = table.query(
            TableName=dynamodb_table,
            IndexName=dynamodb_table_index,
            KeyConditionExpression="InstanceId  = :instance_id",
            ExpressionAttributeValues={
                ":instance_id": instance_id
            }
        )
        tracer.put_annotation(
            key="theater_id", value=response["Items"][0]["TheaterId"])
        tracer.put_annotation(
```

```
            key="theater_name", value=response["Items"][0]
["TheaterName"])
        tracer.put_annotation(
            key="theater_email", value=response["Items"][0]
["TheaterEmail"])
    except botocore.exceptions.ClientError as e:
        return None, json.dumps(e.response)
    return response["Items"], None


# function to fetch command details from SSM run command history using
command id
@tracer.capture_method(capture_response=False)
def get_command_details(command_id, instance_id, ssm_client):
    """
    Fetch command details from SSM run command history using command id.

    Parameters
    ----------
    command_id : str
        Command id.
    instance_id : str
        Instance id.
    ssm_client : boto3.client
        SSM client.

    Returns
    -------
    dict
        SSM command details.
    str
        Error message if command details are not fetched from SSM run
command history.

    """

    try:
        response = ssm_client.get_command_invocation(
            CommandId=command_id,
            InstanceId=instance_id
        )

        tracer.put_annotation(
            key="command_id", value=response["CommandId"])
        tracer.put_annotation(
            key="status", value=response["Status"])
        tracer.put_annotation(
            key="detailed_status", value=response["StatusDetails"])
        tracer.put_annotation(
            key="output", value=response["StandardOutputContent"])
        tracer.put_annotation(
            key="error", value=response["StandardErrorContent"])
    except botocore.exceptions.ClientError as e:
        return None, json.dumps(e.response)
    return response, None
```

```
def download_csv(bucket, key):
    print(key)
    try:
        response = s3_client.get_object(Bucket=bucket, Key=key)
        csv_content = response['Body'].read().decode('utf-8')
        csv_file = io.StringIO(csv_content)
        csv_reader = csv.reader(csv_file)

        # Modify the CSV by extracting only the last folder and the
filename
        modified_csv = []
        for row in csv_reader:
            path_parts = row[0].split('/')
            if len(path_parts) >= 2:
                filename = path_parts[-2] + '/' + path_parts[-1]
            else:
                filename = row[0]
            modified_row = [filename, row[1]]
            modified_csv.append(modified_row)

        return modified_csv
    except botocore.exceptions.ClientError as e:
        logger.error(e)
        return None, json.dumps(e.response)


# function to send notification to the user using SNS topic


@tracer.capture_method(capture_response=False)
def send_notification(email, theaterName, command_details, message,
theater_id):
    """
    Send notification to the user using SES.

    Parameters
    ----------
    email : str
        Email address.

    theaterName : str
        Theater name.

    command_details : dict
        SSM command details.

    message : dict
        SSM command message.

    Returns
    -------
    dict
        SNS response.
    str
        Error message if notification is not sent.

    """
```

```
    folder_name, info, start_time, end_time, current_time, status,
command_id, output, error, source_path, dest_path, source_hash,
dest_hash = clean_data(
        command_details, message)
    # extract date from start time in format YYYY-MM-DD
    date = start_time.split(' ')[0]
    template = None
    client_csv = None
    server_csv = None
    if status == 'Success':
        template = env.get_template('template/
email_template_success.html')

        # Download the client and server CSV files from S3
        client_csv = download_csv(s3_bucket, source_hash)
        server_csv = download_csv(s3_bucket, dest_hash)

        logMsg = "Success on " + theaterName + "  " + info
        logger.append_keys(command_id=command_id)
        logger.append_keys(theaterName=theaterName)
        logger.append_keys(status=status)
        logger.append_keys(source_path=source_path)
        logger.append_keys(dest_path=dest_path)
        logger.info(logMsg)

    else:
        template = env.get_template('template/email_template.html')

        logger.append_keys(command_id=command_id)
        logger.append_keys(theaterName=theaterName)
        logger.append_keys(status=status)
    html = template.render(theaterName=theaterName,
folder_name=folder_name, status=status, info=info, error=error,
                           command_id=command_id, start_time=start_time,
end_time=end_time, current_time=current_time)

    # Create a MIME multipart message object
    msg = MIMEMultipart()
    html_part = MIMEText(html, 'html')
    msg.attach(html_part)

    # Add the first image as an attachment to the message object
    img_part1 = MIMEImage(img_data1, name='logo.png')
    img_part1.add_header('Content-Disposition',
                         'attachment', filename='logo.png')
    img_part1.add_header('Content-ID', '')
    msg.attach(img_part1)

    # Add the second image as an attachment to the message object
    img_part2 = MIMEImage(img_data2, name='header.png')
    img_part2.add_header('Content-Disposition',
                         'attachment', filename='header.png')
    img_part2.add_header('Content-ID', '
')
    msg.attach(img_part2)
```

```
    # attach both csv files to the email
    if client_csv is not None and server_csv is not None:
        # join rows with ',' and lines with '\r'
        client_csv_str = '\r'.join([','.join(row) for row in
client_csv])
        server_csv_str = '\r'.join([','.join(row) for row in
server_csv])

        # create MIMEText objects with the CSV data
        csv_part1 = MIMEText(client_csv_str, _subtype='csv')
        csv_part1.add_header('Content-Disposition',
                             'attachment', filename='client-hash.csv')

        csv_part2 = MIMEText(server_csv_str, _subtype='csv')
        csv_part2.add_header('Content-Disposition',
                             'attachment', filename='server-hash.csv')

        # attach MIMEText objects to the email message
        msg.attach(csv_part1)
        msg.attach(csv_part2)

    subject = "DCP Delivery Status for " + theaterName + " - " + status
    msg['Subject'] = subject
    msg['From'] = "DCinema Distribution Status <" + emailfrom + ">"
    msg['To'] = email

    try:
        response = ses_client.send_raw_email(
            Source=emailfrom,
            Destinations=[email],
            RawMessage={'Data': msg.as_string()}
        )
    except botocore.exceptions.ClientError as e:
        return None, json.dumps(e.response)
    return response, None


# SES , create new using the environment variable EMAIL_FROM if not
present. Else, use the existing one.
@tracer.capture_method(capture_response=False)
def verify_email_identity(email_from, ses_client):
    """
    SES , create new using the environment variable EMAIL_FROM if not
present. Else, use the existing one.

    Parameters
    ----------
    email_from : str
        Email address.

    ses_client : boto3.client
        SES client.

    Returns
    -------
    str
        Email address.
```

```
    str
        Error message if email address is not verified.

    """

    try:
        response = ses_client.list_identities(
            IdentityType='EmailAddress',
            MaxItems=123
        )
        # check if the email address is already exists in the SES, if
not then create new one using the environment variable EMAIL_FROM, else
use the existing one return email from.
        if email_from not in response["Identities"]:
            response = ses_client.verify_email_identity(
                EmailAddress=email_from
            )
            # wait for the email to be verified
            ses_client.get_identity_verification_attributes(
                Identities=[email_from]
            )
            return email_from, None
        else:
            return email_from, None
    except botocore.exceptions.ClientError as e:
        return None, json.dumps(e.response)


# lambda handler function
@tracer.capture_lambda_handler
def lambda_handler(event, context):
    """
    Lambda handler function.

    Parameters
    ----------
    event : dict
        Event data.

    context : object
        Lambda Context runtime methods and attributes.

    Returns
    -------
    dict
        Response.
    str
        Error message if any.

    """

    # fetch instance id from event
    # fetch command id from event which is an SNSEvent, the body of the
event is a json string{"commandId":"7463d40e-a888-4a09-96da-
ed0f42773554","documentName":"DCinema-SSMDocument-
tNyLEG2ZHRZy","instanceId":"mi-076b2425febe0914e","requestedDateTime":"2
023-02-20T13:22:47.378Z","status":"InProgress","detailedStatus":"InProgr
```

```
ess","eventTime":"2023-02-20T13:22:47.406Z"}
    message = json.loads(event["Records"][0]["Sns"]["Message"])
    instance_id = message["instanceId"]
    command_id = message["commandId"]
    # add to dynamodb table

    # verify email from SES
    email_from, error = verify_email_identity(emailfrom, ses_client)
    if error:
        logger.error(error)
        return None, error

    # fetch theater details from dynamodb table using instance id
    theater_details, error = get_instance_details(instance_id, table)
    if error:
        logger.error(error)
        return None, error

    # use command id to fetch command details from SSM run command
history
    command_details, error = get_command_details(
        command_id, instance_id, ssm_client)
    if error:
        logger.error(error)
        return None, error
    # store the command details in dynamodb table

    # use theater email id to send notification to the user, with the
status of the SSM document execution output
    theaterName = theater_details[0]["TheaterName"]
    email = theater_details[0]["TheaterEmail"]
    theater_id = theater_details[0]["TheaterId"]
    res, error = store_status_in_dynamodb(command_details, theater_id,
message)
    if error:
        logger.error(error)
        return None, error
    response, error = send_notification(
        email, theaterName, command_details, message, theater_id)
    if error:
        logger.error(error)
        return None, error
    return response, None
```

ValidateTransfer

```
import boto3
import csv
import io
import os
import datetime
import json
from aws_lambda_powertools import Logger
from aws_lambda_powertools import Tracer

logger = Logger()
tracer = Tracer()
s3 = boto3.client('s3')
ssm = boto3.client('ssm')

client_bucket = os.environ['S3_BUCKET']
server_bucket = os.environ['S3_BUCKET']


def compare_hashes(file1_hashes, file2_hashes):
    # Compare the hashes and output any mismatches
    matches = 0
    non_matches = 0
    for filename in set(file1_hashes.keys()) | set(file2_hashes.keys()):
        if filename not in file1_hashes:
            print(f"{filename} not found in Client")
            non_matches += 1
            continue
        if filename not in file2_hashes:
            print(f"{filename} not found in Server")
            non_matches += 1
            continue
        if file1_hashes[filename] != file2_hashes[filename]:
            print(f"Hash mismatch for {filename}")
            non_matches += 1
            continue
        matches += 1

    print(f"Found {matches} matching files and {non_matches} non-
matching files")

    result = {
        "matches": matches,
        "non_matches": non_matches
    }
    return result


def download_csv(bucket, key):
    response = s3.get_object(Bucket=bucket, Key=key)
    csv_content = response['Body'].read().decode('utf-8')
    csv_file = io.StringIO(csv_content)
    csv_reader = csv.reader(csv_file)
```

```
    return list(csv_reader)


def lambda_handler(event, context):
    # Get the S3 bucket and key for the client and server CSV files
    path = event['path']
    body = json.loads(event['body'])
    client_id = body['client_id']
    date = body['date']

    try:
        folder_name = body['folder_name']
        server_key = 'FolderIntegrity/Server/{date}/{folder_name}/
source.csv'.format(
            date=date,
            folder_name=folder_name
        )
        client_key = body['client_key']
        # Download the client and server CSV files from S3
        client_csv = download_csv(client_bucket, client_key)
        server_csv = download_csv(server_bucket, server_key)

        # Convert the CSV data to dictionaries of hashes
        client_hashes = {}
        for row in client_csv[1:]:
            filename = os.path.basename(row[0])
            client_hashes[filename] = row[1]

        server_hashes = {}
        for row in server_csv[1:]:
            filename = os.path.basename(row[0])
            server_hashes[filename] = row[1]

        # Compare the hashes and output any mismatches
        result = compare_hashes(client_hashes, server_hashes)
        # add server_key and client_key to result
        result['server_key'] = server_key
        result['client_key'] = client_key
        return {
            'statusCode': 200,
            'body': json.dumps(result)
        }
    except Exception as e:
        print(e)
        return {
            'statusCode': 500,
            'body': 'Failed to compare hashes'
        }
```