Problem Statement:

AcmeRetail’s monthly AWS bill is up 28% while EC2 CPU sits mostly in single digits. Engineers open CloudWatch dashboards ad-hoc and finance relies on Cost Explorer once a week, so no one gets daily, prioritized actions. We will build a small serverless service that:

* Pulls the last 7 days of UnblendedCost (grouped by service) from Cost Explorer.
* Reads per-instance CPU for the last hour from CloudWatch and EC2 metadata.
* Summarizes risks and concrete savings recommendations using an LLM on Amazon Bedrock.
* Persists every analysis with latency and model meta into DynamoDB.
* Exposes a simple HTTPS endpoint (Lambda Function URL) so anyone—or ChatGPT—can ask questions like “Where can we rightsize today?”

**Success criteria**

* One POST call returns a human-readable summary plus latency in milliseconds.
* A DynamoDB item is written for each call.
* Team can run it with 512 MB memory, 5-minute timeout, and (for the quickstart) admin permissions on the execution role, then tighten later.

**2.2 Prerequisites**

* AWS account with permissions to create IAM roles/policies, Lambda, DynamoDB, and enable Bedrock model access.
* Cost Explorer enabled for the account.
* Bedrock access to the chosen model ID (e.g., us.anthropic.claude-3-haiku-20240307-v1:0) in your selected region.
* Basic familiarity with the AWS Console or AWS CLI.
* Your Lambda code (provided in the prompt).

Note on regions

* Keep all services in the same region for simplicity. If your chosen Bedrock model is not available in that region, use a supported region for Bedrock and set the Lambda env vars accordingly. Your code currently defaults to us-east-2; set AWS\_REGION to match your plan.

**2.3 Approach to solve it**

1. Create a DynamoDB table to store every analysis (pk + ts as keys).
2. Create a Lambda function (Python 3.x) using the provided code.
3. Configure environment variables: TABLE, BEDROCK\_MODEL, and AWS\_REGION (or your preferred).
4. Grant the function permissions for Cost Explorer, EC2, CloudWatch, DynamoDB, Bedrock, and logs.
5. Increase Lambda timeout to 5 minutes and memory to 512 MB to accommodate LLM latency.
6. Create a Function URL to invoke the service over HTTPS.
7. Test with curl/postman, verify DynamoDB writes, and confirm a readable LLM summary is returned.

**Project Solution**

**2.4 Complete setup guide**

Step 0 — Enable Cost Explorer and Bedrock model access (only if its not enabled)

* In Billing > Cost Explorer, enable if it’s not already on.
* In Bedrock console, enable access to your target model (e.g., Claude 3 Haiku) in your region.



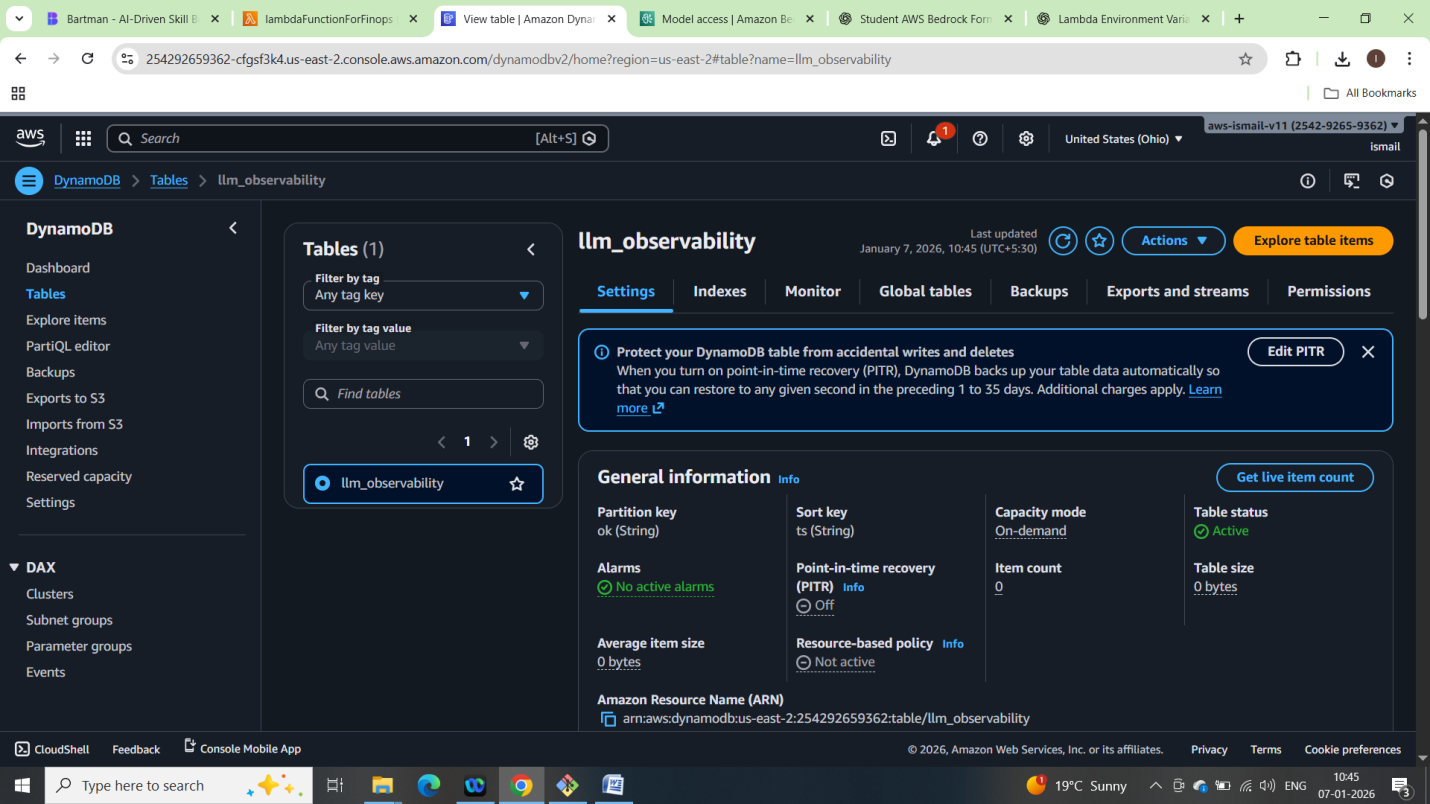
Amazon Bedrock model acces is enabled by default



Step 1 — Create DynamoDB table

* Name: llm\_observability (or your choice—match the env var).
* Partition key: pk (String).
* Sort key: ts (Number).
* Billing mode: On-demand is fine for this project.
* No GSIs needed for now.

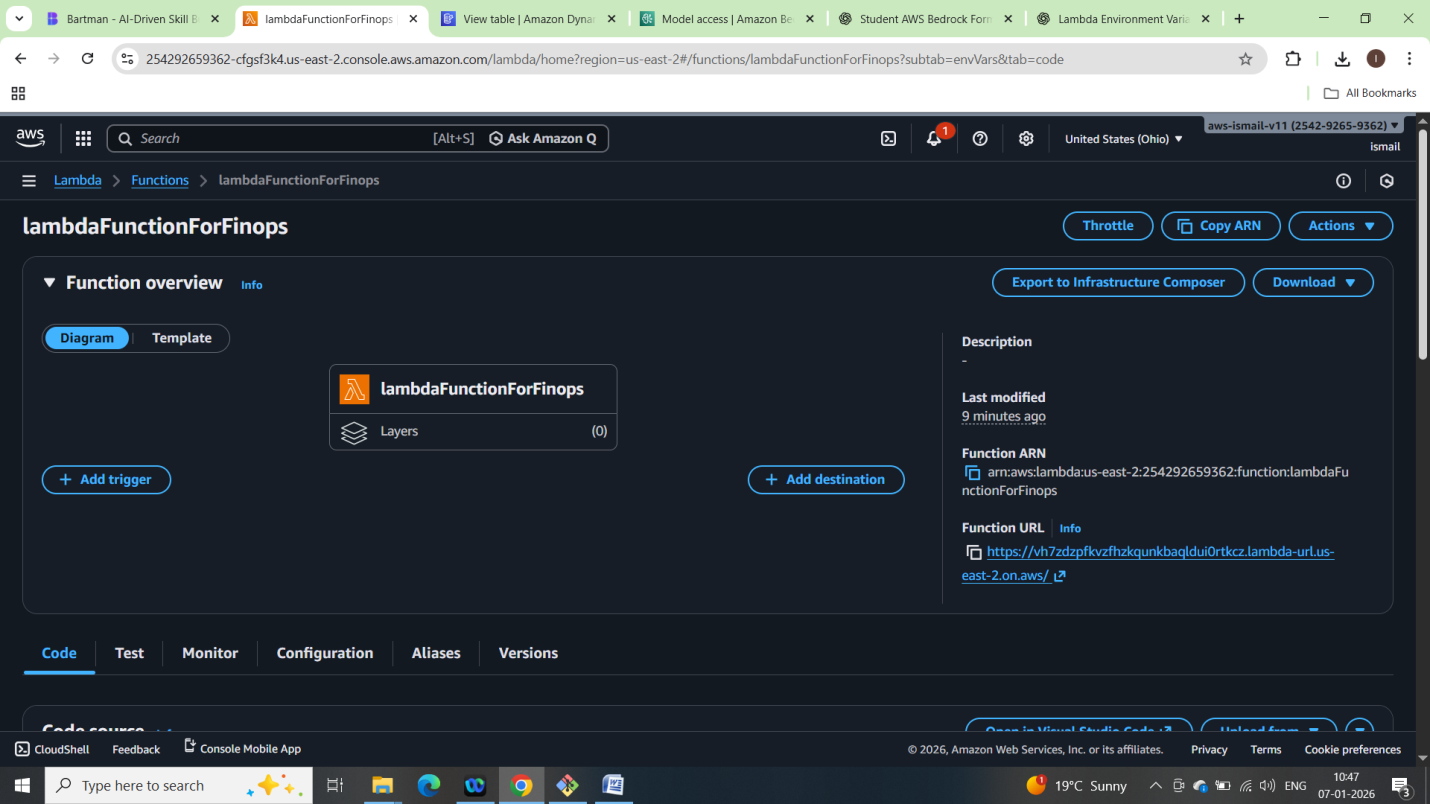
Created llm\_observablility table in dynamodb



Step 2 — Create the Lambda function

* Runtime: Python 3.12 (or 3.11).
* Architecture: x86\_64 is fine.
* Handler: lambda\_function.handler (or your filename).
* Upload your Python file as lambda\_function.py (or zip if using layers/deps; this code uses only the built-in boto3).

Created lambda function named lambdaFunctionForFinops as per above



Code:

# lambda\_function.py

import boto3, json, os, time, uuid, logging

from datetime import datetime, timedelta, timezone

from decimal import Decimal

from botocore.config import Config

# -------- Settings --------

REGION = os.environ.get("AWS\_REGION", "us-east-2")

MODEL\_ID = os.environ.get("BEDROCK\_MODEL", "us.anthropic.claude-3-haiku-20240307-v1:0")

TABLE\_NAME = os.environ["TABLE"]  # fail early if missing

# Optional knobs

CPU\_LOOKBACK\_MIN = int(os.environ.get("CPU\_LOOKBACK\_MIN", "60"))      # last 60 mins

CPU\_PERIOD\_SEC   = int(os.environ.get("CPU\_PERIOD\_SEC", "300"))       # 5 min bins

COST\_LOOKBACK\_DAYS = int(os.environ.get("COST\_LOOKBACK\_DAYS", "7"))

REQUIRE\_API\_KEY = None

TEMPERATURE = float(os.environ.get("TEMPERATURE", "0.3"))

MAX\_TOKENS = int(os.environ.get("MAX\_TOKENS", "600"))

# boto config: small retries

boto\_cfg = Config(retries={"max\_attempts": 4, "mode": "standard"})

log = logging.getLogger()

log.setLevel(logging.INFO)

cloudwatch = boto3.client("cloudwatch", region\_name=REGION, config=boto\_cfg)

ec2 = boto3.client("ec2", region\_name=REGION, config=boto\_cfg)

ddb = boto3.resource("dynamodb", region\_name=REGION).Table(TABLE\_NAME)

ce = boto3.client("ce", config=boto\_cfg)

bedrock = boto3.client("bedrock-runtime", region\_name=REGION, config=boto\_cfg)

def \_utcnow():

    return datetime.now(timezone.utc)

def get\_cost\_summary():

    end = \_utcnow().strftime("%Y-%m-%d")

    start = (\_utcnow() - timedelta(days=COST\_LOOKBACK\_DAYS)).strftime("%Y-%m-%d")

    token = None

    lines = []

    while True:

        resp = ce.get\_cost\_and\_usage(

            TimePeriod={"Start": start, "End": end},

            Granularity="DAILY",

            Metrics=["UnblendedCost"],

            GroupBy=[{"Type": "DIMENSION", "Key": "SERVICE"}],

            NextPageToken=token

        ) if token else ce.get\_cost\_and\_usage(

            TimePeriod={"Start": start, "End": end},

            Granularity="DAILY",

            Metrics=["UnblendedCost"],

            GroupBy=[{"Type": "DIMENSION", "Key": "SERVICE"}]

        )

        for day in resp.get("ResultsByTime", []):

            for g in day.get("Groups", []):

                service = g["Keys"][0]

                amount = g["Metrics"]["UnblendedCost"]["Amount"]

                try:

                    lines.append(f"{service}: ${float(amount):.2f}")

                except Exception:

                    # handle "-" or missing values

                    lines.append(f"{service}: ${amount}")

        token = resp.get("NextPageToken")

        if not token:

            break

    if not lines:

        return "No cost data returned for the selected window."

    # Keep it terse: last day plus a few others can be overwhelming;

    # we’ll just join everything; the LLM can handle summarizing.

    return "\n".join(lines)

def to\_ddb(value):

    if isinstance(value, float):

        return Decimal(str(value))

    if isinstance(value, dict):

        return {k: to\_ddb(v) for k, v in value.items()}

    if isinstance(value, list):

        return [to\_ddb(v) for v in value]

    return value

def get\_cpu\_average(instance\_id):

    end = \_utcnow()

    start = end - timedelta(minutes=CPU\_LOOKBACK\_MIN)

    stats = cloudwatch.get\_metric\_statistics(

        Namespace="AWS/EC2",

        MetricName="CPUUtilization",

        Dimensions=[{"Name": "InstanceId", "Value": instance\_id}],

        StartTime=start,

        EndTime=end,

        Period=CPU\_PERIOD\_SEC,

        Statistics=["Average"]

    )

    dps = stats.get("Datapoints", [])

    if not dps:

        return 0.0

    return round(sum(d["Average"] for d in dps) / len(dps), 2)

def summarize\_with\_llm(context\_text):

    prompt = (

        "You are an expert AWS DevOps & FinOps assistant. "

        "From the cost and utilization context, produce a short executive summary, "

        "then 3-6 prioritized, low-risk savings or hygiene actions. "

        "Call out anomalies, specify instance IDs where relevant, and note verification steps.\n\n"

        f"{context\_text}"

    )

    body = {

        "anthropic\_version": "bedrock-2023-05-31",

        "max\_tokens": MAX\_TOKENS,

        "temperature": TEMPERATURE,

        "messages": [

            {"role": "user", "content": [{"type": "text", "text": prompt}]}

        ]

    }

    resp = bedrock.invoke\_model(modelId=MODEL\_ID, body=json.dumps(body))

    result = json.loads(resp["body"].read())

    return result.get("content", [{}])[0].get("text", json.dumps(result))

def \_paginate\_instances():

    token = None

    while True:

        kwargs = {"NextToken": token} if token else {}

        page = ec2.describe\_instances(\*\*kwargs)

        for r in page.get("Reservations", []):

            for i in r.get("Instances", []):

                yield i

        token = page.get("NextToken")

        if not token:

            break

def \_cors\_headers():

    return {

        "Access-Control-Allow-Origin": "\*",

        "Access-Control-Allow-Methods": "POST, OPTIONS",

        "Access-Control-Allow-Headers": "Content-Type, X-Api-Key"

    }

def lambda\_handler(event, context):

    try:

        if event.get("httpMethod") == "OPTIONS":

            return {"statusCode": 200, "headers": \_cors\_headers(), "body": ""}

        # Optional shared-secret check for Function URL

        if REQUIRE\_API\_KEY:

            hdr = (event.get("headers") or {}).get("x-api-key") or (event.get("headers") or {}).get("X-Api-Key")

            if hdr != REQUIRE\_API\_KEY:

                return {

                    "statusCode": 401,

                    "headers": \_cors\_headers(),

                    "body": json.dumps({"error": "unauthorized"})

                }

        raw\_body = event.get("body", "") or ""

        body = json.loads(raw\_body) if raw\_body and raw\_body.strip().startswith("{") else {}

        question = body.get("question", "Analyze EC2 utilization trends for the last hour.")

        if not isinstance(question, str):

            question = str(question)

        if len(question) > 2000:

            question = question[:2000]

        log.info(json.dumps({"msg": "request\_received", "len\_question": len(question)}))

        # Build utilization report

        report\_lines = []

        for inst in \_paginate\_instances():

            iid = inst["InstanceId"]

            name = next((t["Value"] for t in inst.get("Tags", []) if t["Key"] == "Name"), iid)

            state = inst["State"]["Name"]

            cpu = get\_cpu\_average(iid)

            report\_lines.append(f"{name} ({iid}) - state={state}, avgCPU={cpu}%")

        context\_text = get\_cost\_summary() + "\n\n" + "\n".join(report\_lines) + f"\n\nQuestion: {question}"

        start = time.time()

        answer = summarize\_with\_llm(context\_text)

        latency = int((time.time() - start) \* 1000)

        item = to\_ddb({

            "pk": str(uuid.uuid4()),

            "ts": int(time.time()),

            "query": question,

            "analysis": answer,

            "latencyMs": latency,

            "model": MODEL\_ID

        })

        ddb.put\_item(Item=item)

        log.info(json.dumps({"msg": "request\_complete", "latencyMs": latency, "items\_written": 1}))

        return {

            "statusCode": 200,

            "headers": {"Content-Type": "application/json", \*\*\_cors\_headers()},

            "body": json.dumps({"answer": answer, "latencyMs": latency, "model": MODEL\_ID})

        }

    except Exception as e:

        log.exception("handler\_error")

        return {

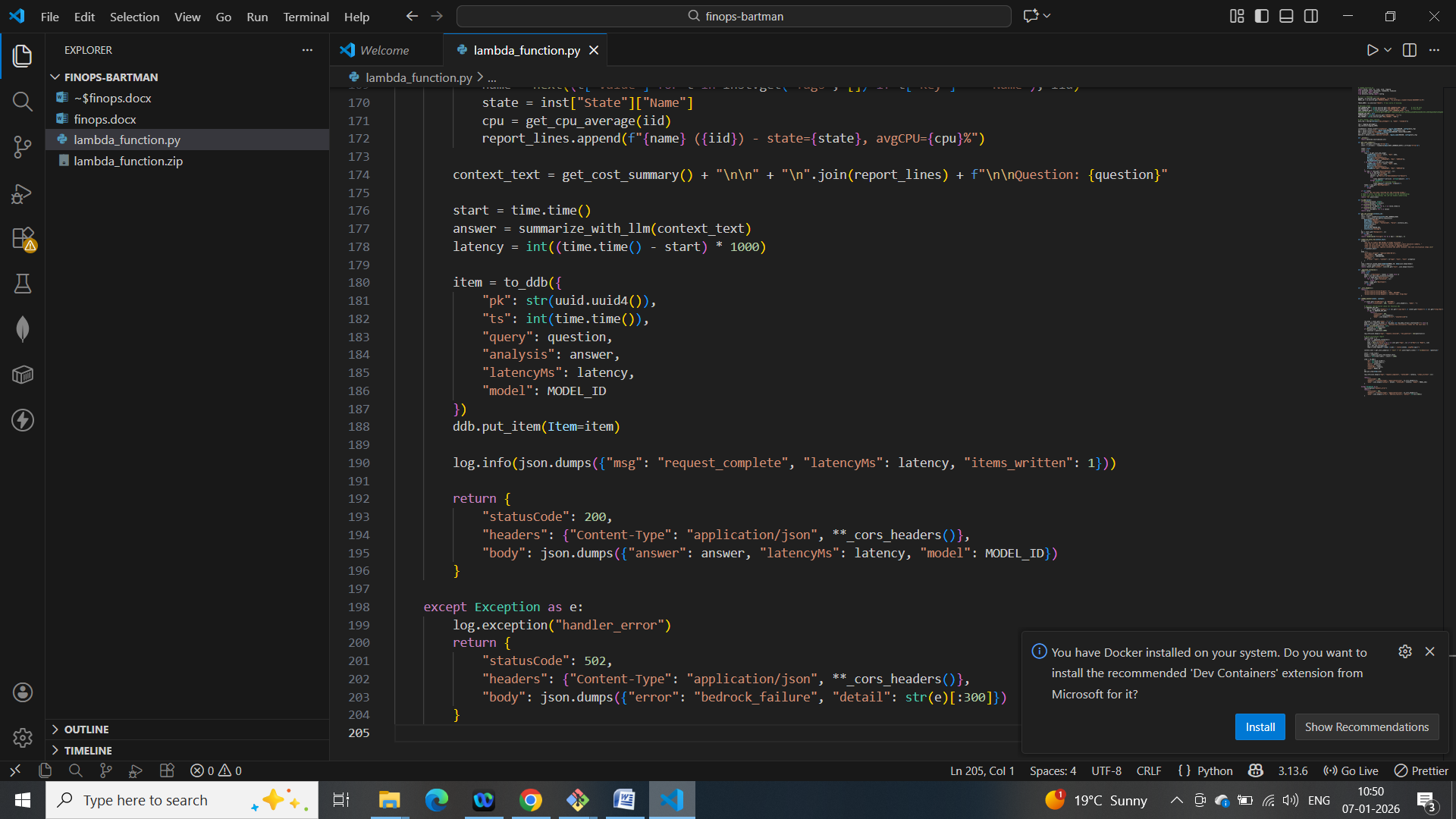
            "statusCode": 502,

            "headers": {"Content-Type": "application/json", \*\*\_cors\_headers()},

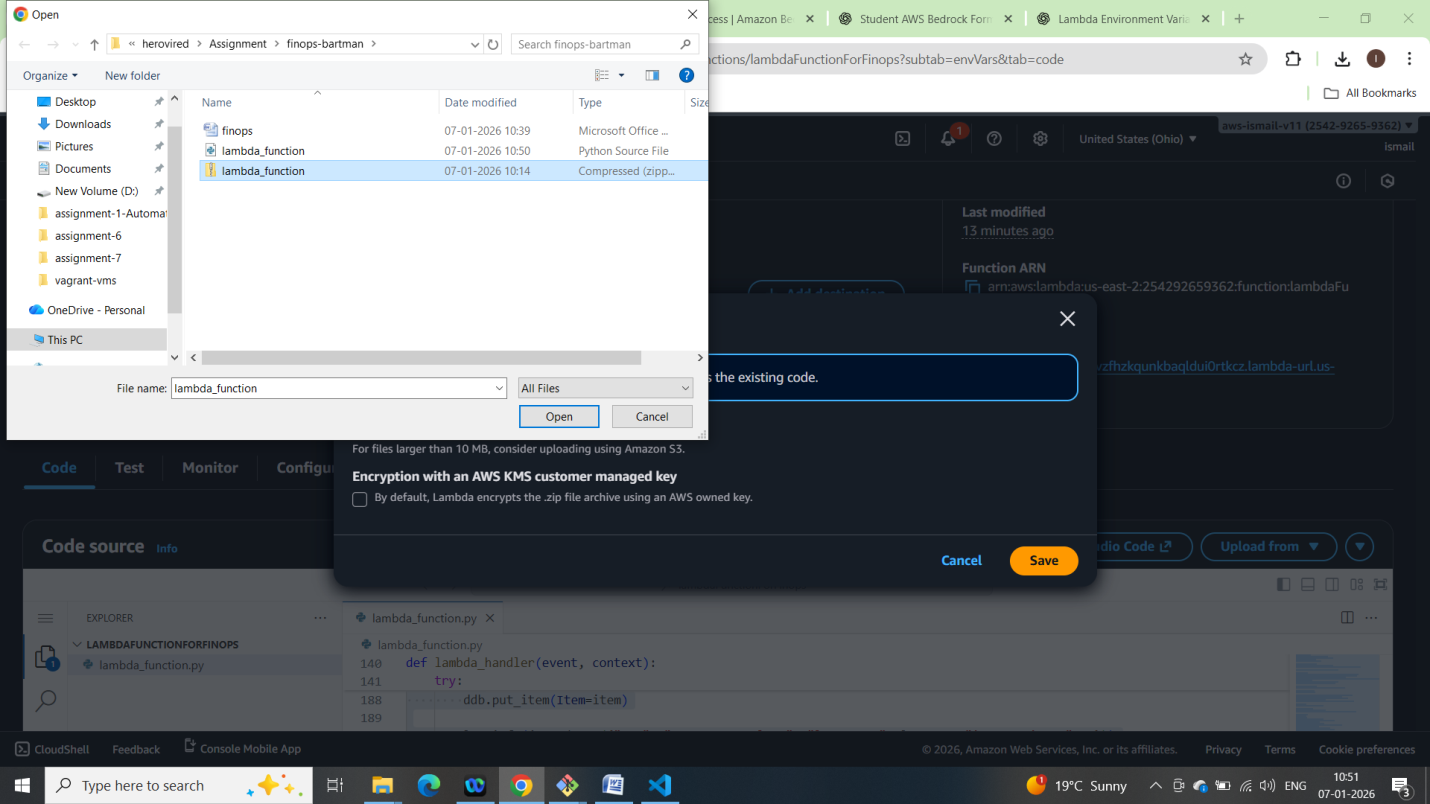
            "body": json.dumps({"error": "bedrock\_failure", "detail": str(e)[:300]})

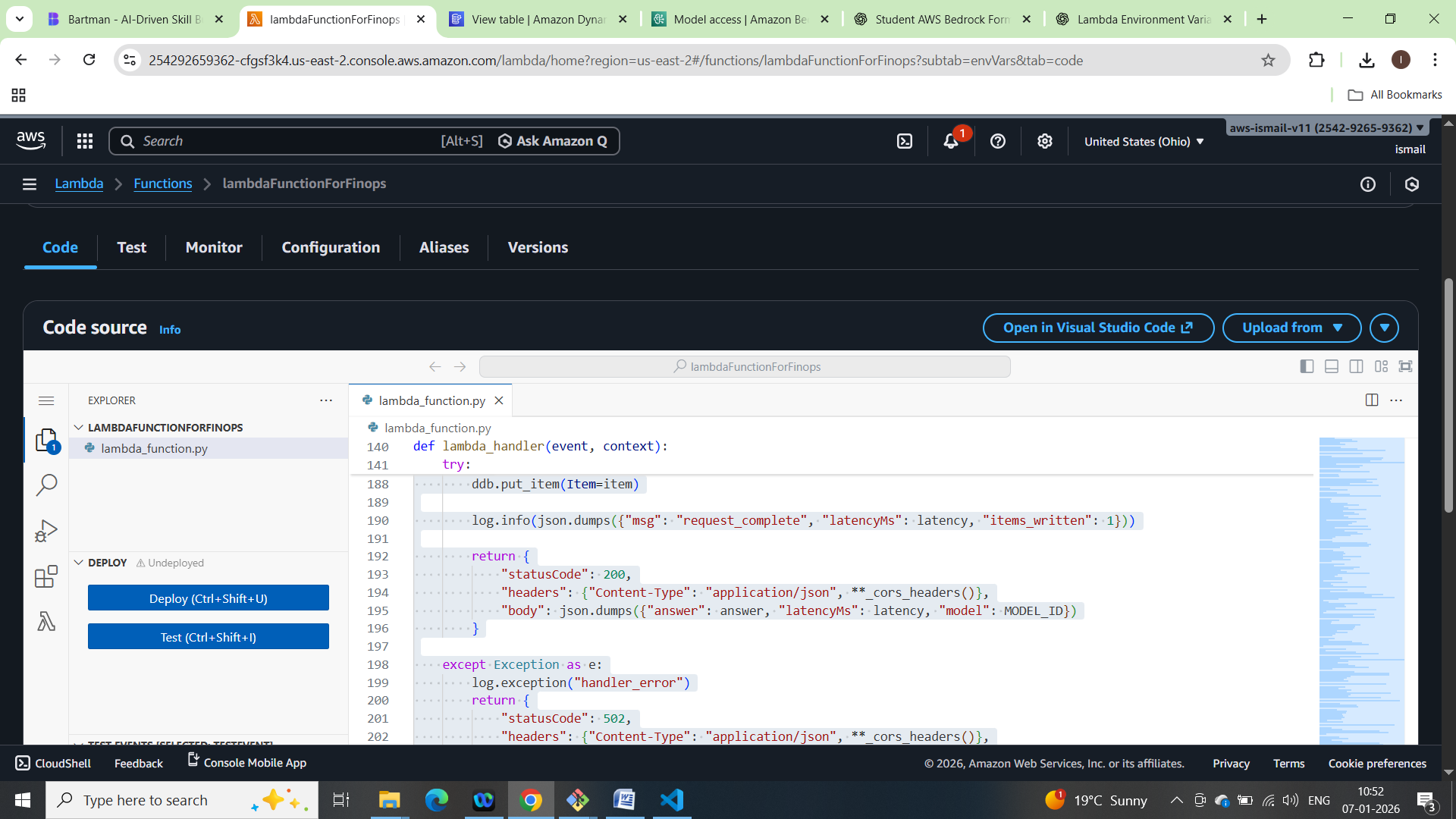
        }

Creted an lambda\_function.py with above code



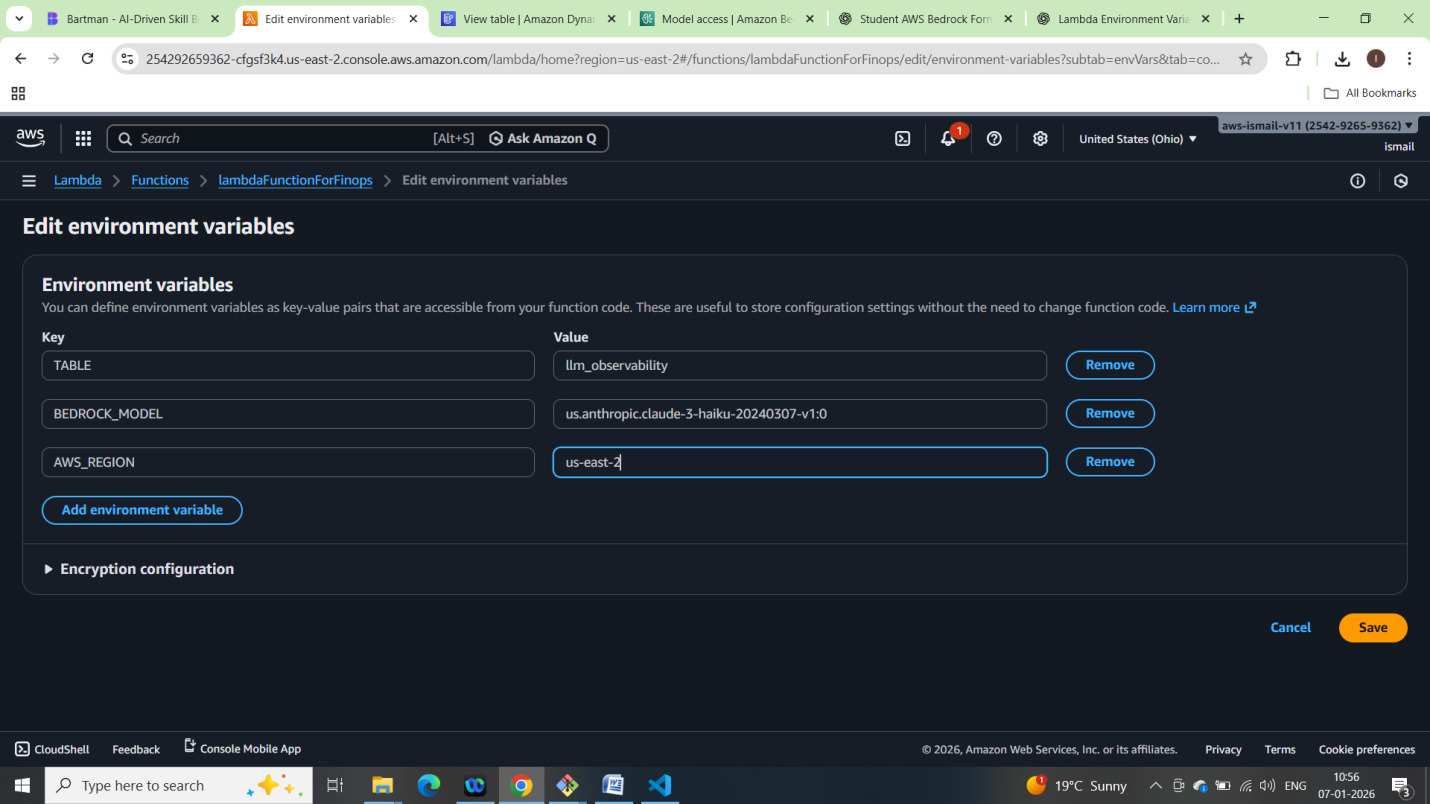
Made a zip file and uploaded on lambda function:





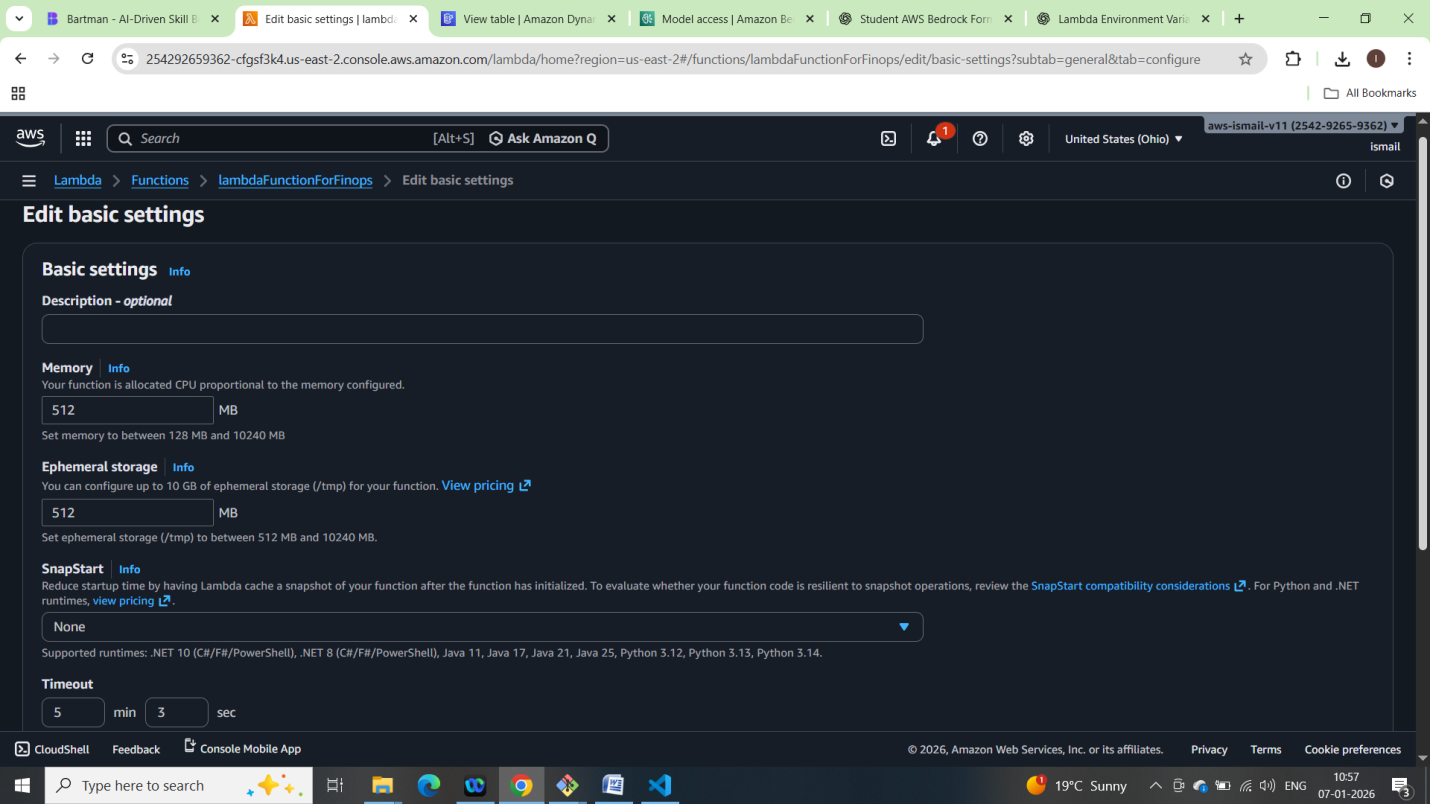
Step 3 — Set environment variables

* TABLE = llm\_observability
* BEDROCK\_MODEL = us.anthropic.claude-3-haiku-20240307-v1:0
* AWS\_REGION = us-east-2 (or the region you picked)
* Optional (if you prefer): also set BEDROCK\_REGION for clarity, but your code reads AWS\_REGION.



Step 4 — Configure memory and timeout

* Memory: 512 MB
* Timeout: 5 minutes (300 seconds)



Step 5 — Execution role and permissions

Quickstart (what you requested):

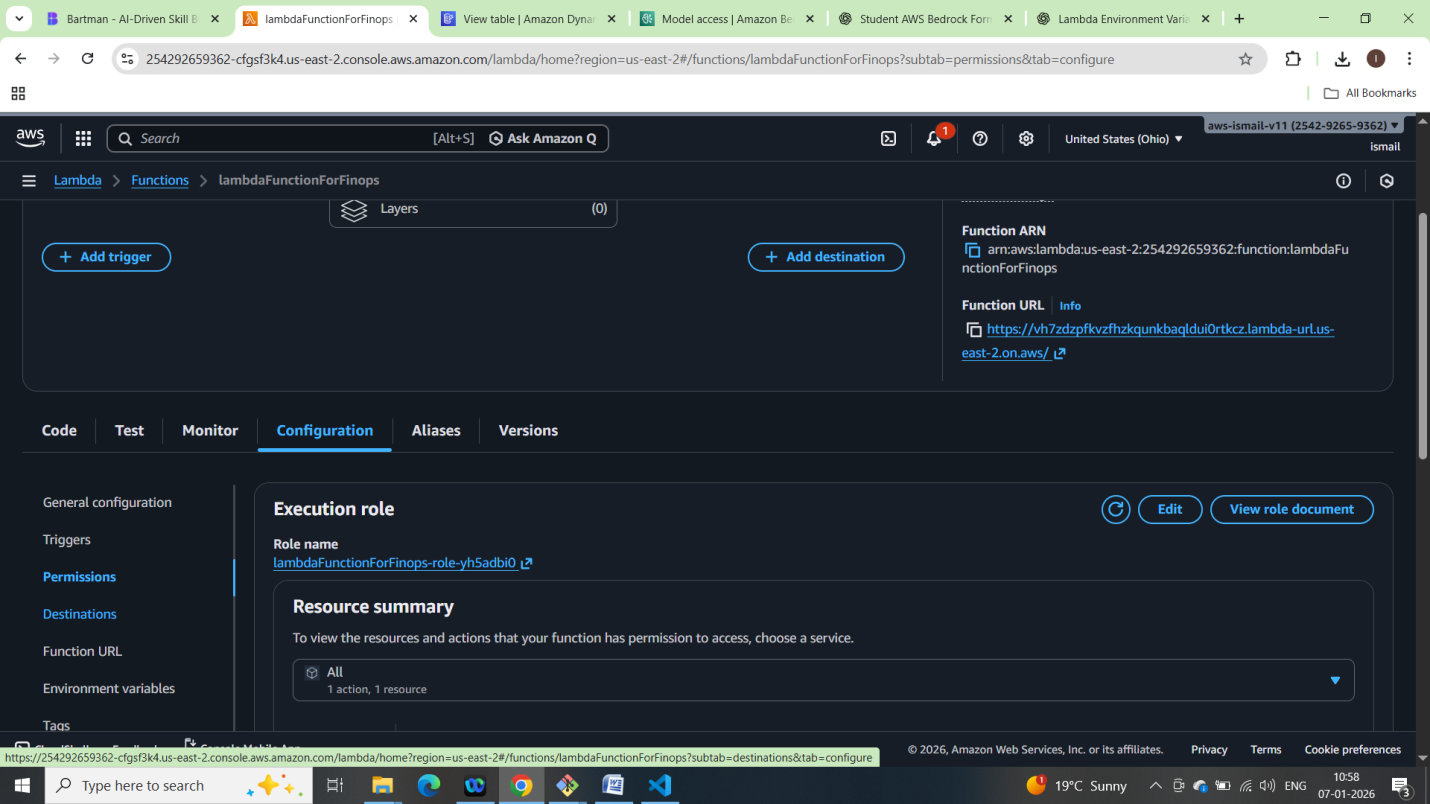
* Attach AdministratorAccess to the Lambda execution role to avoid permission blockers during the first run.

*Best practice (apply after initial validation):*

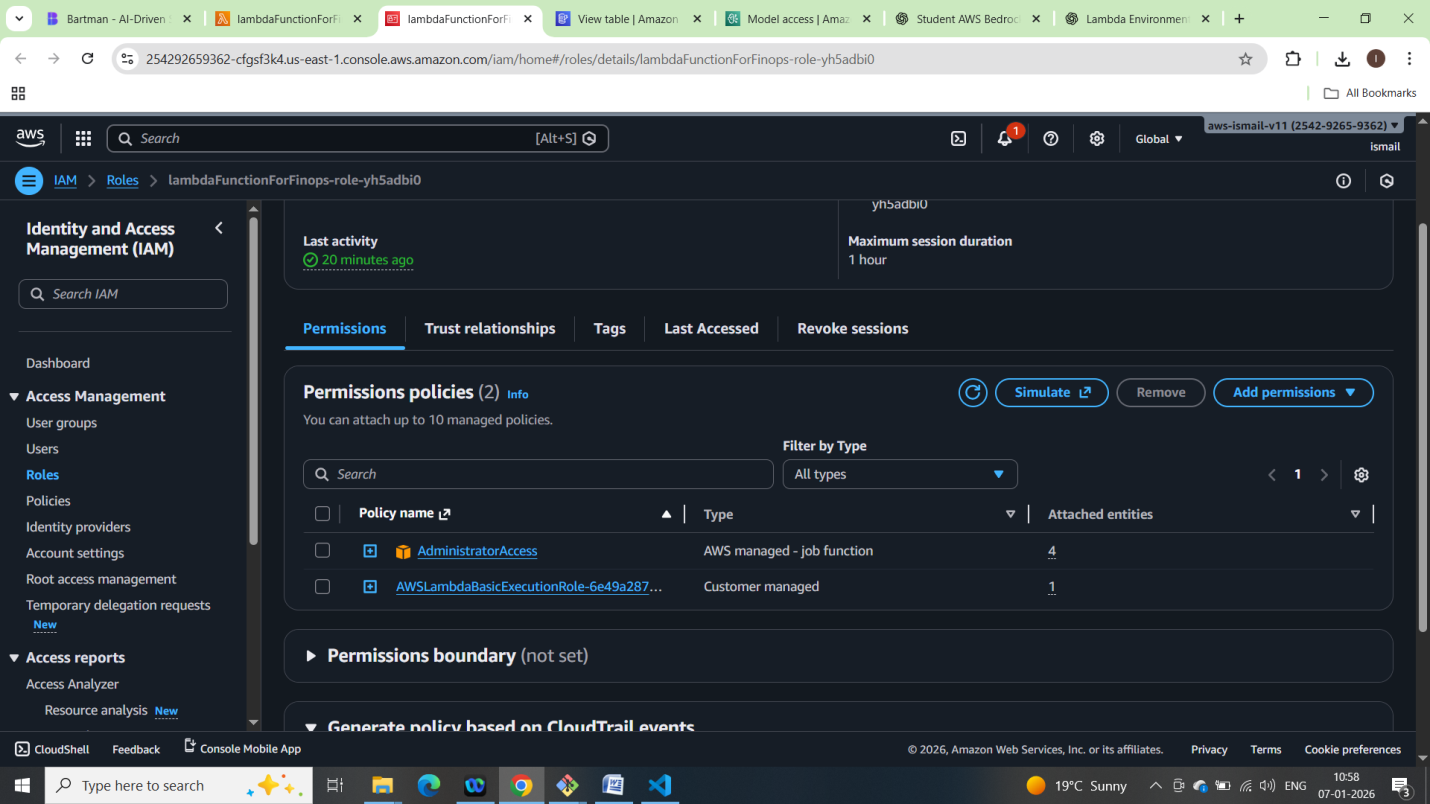
Replace admin with a least-privilege inline policy that covers exactly what the function uses. Example policy actions (tighten resources later):

IAM policy actions (allow on Resource: "\*", then scope down):

* Logs
* logs:CreateLogGroup, logs:CreateLogStream, logs:PutLogEvents
* Cost Explorer
* ce:GetCostAndUsage
* EC2 and CloudWatch metrics
* ec2:DescribeInstances
* cloudwatch:GetMetricStatistics
* DynamoDB
* dynamodb:PutItem, dynamodb:DescribeTable
* Bedrock
* bedrock:InvokeModel, bedrock:InvokeModelWithResponseStream

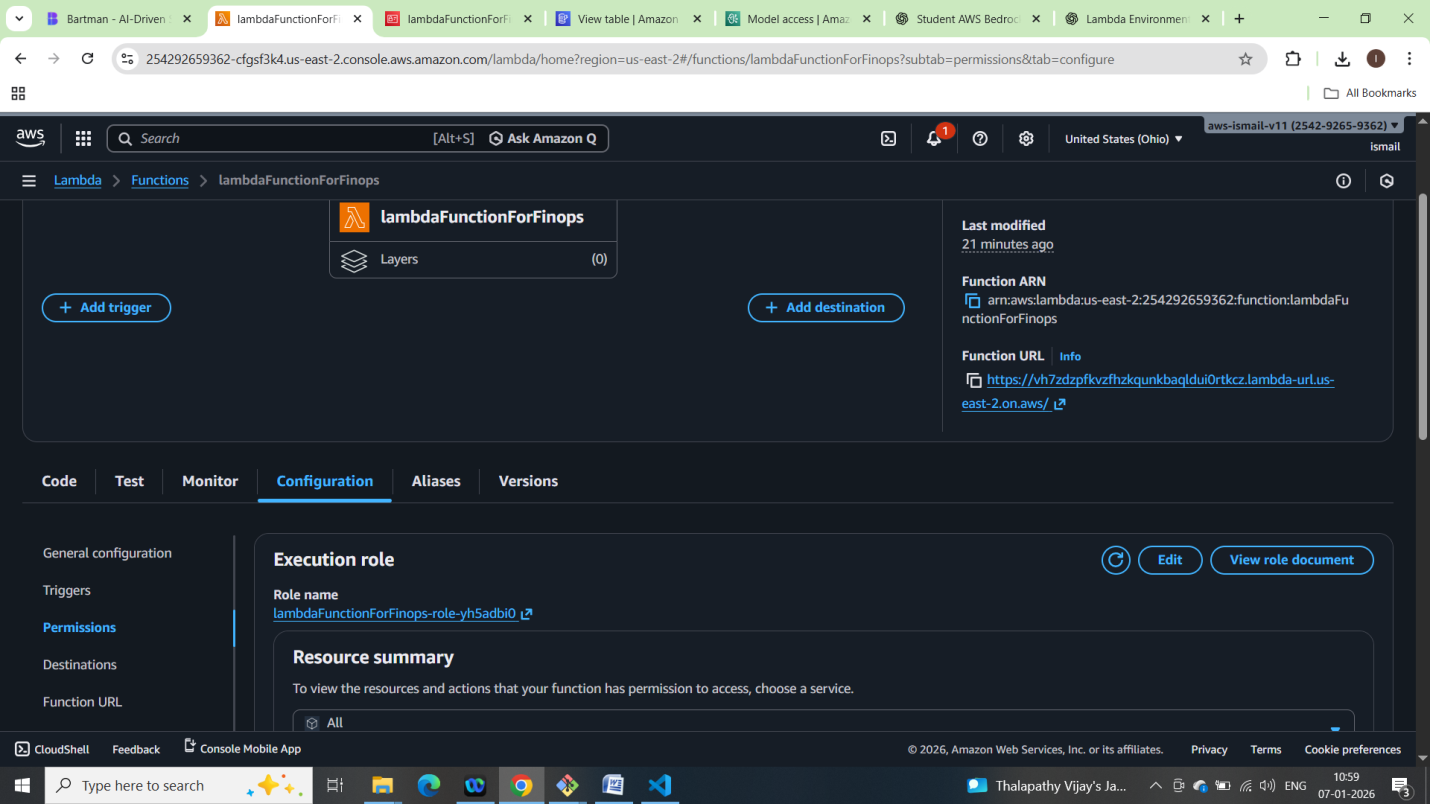


Provided administrator access



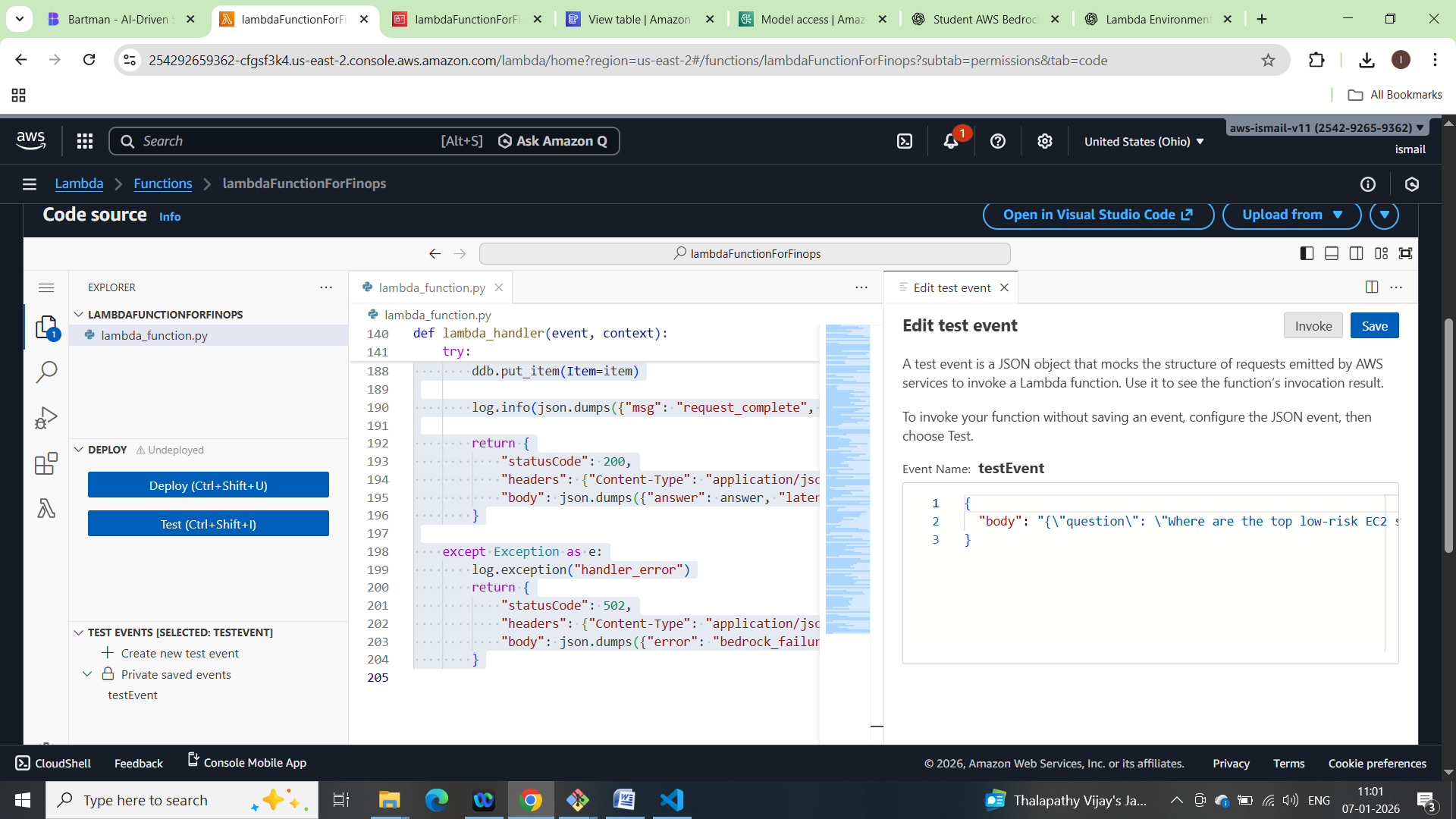
Step 6 — Create a Function URL

* In Lambda > your function > Function URL > Create.
* Auth type: choose NONE for simplest testing or AWS\_IAM if you want signed requests.
* If you choose NONE, consider adding an API key header check in code later.



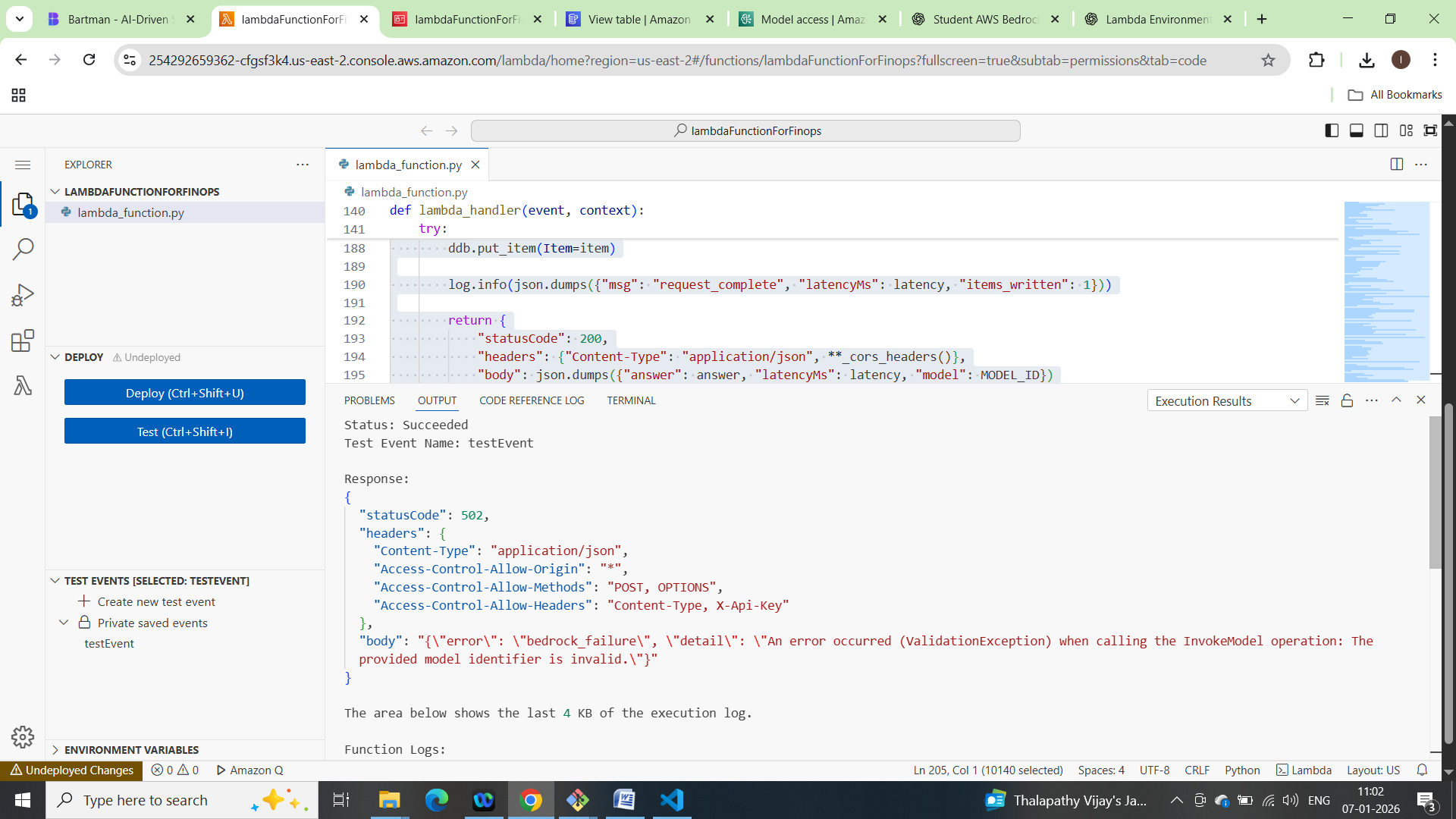
Testing the lambda function:

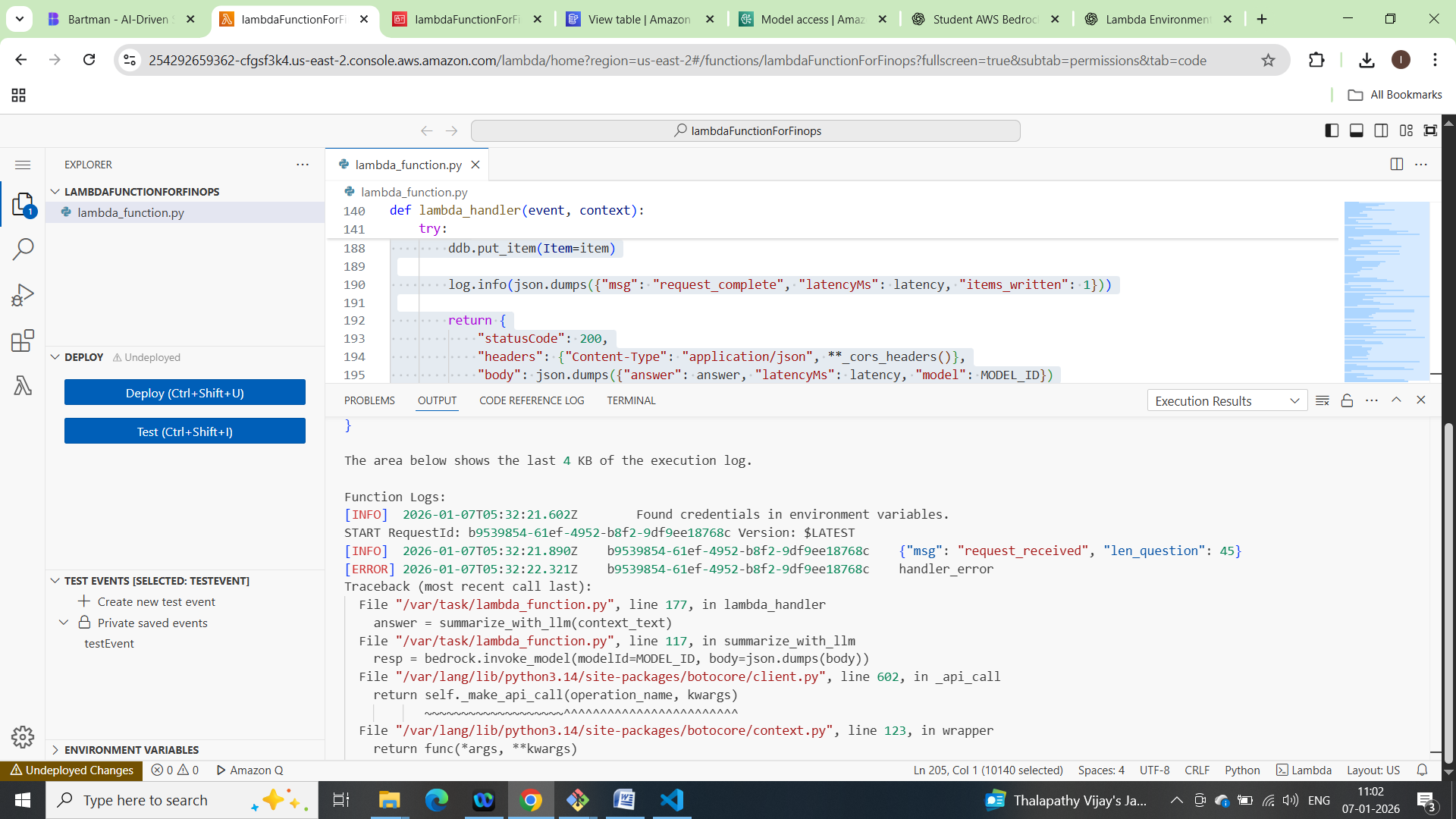
Created testEvent



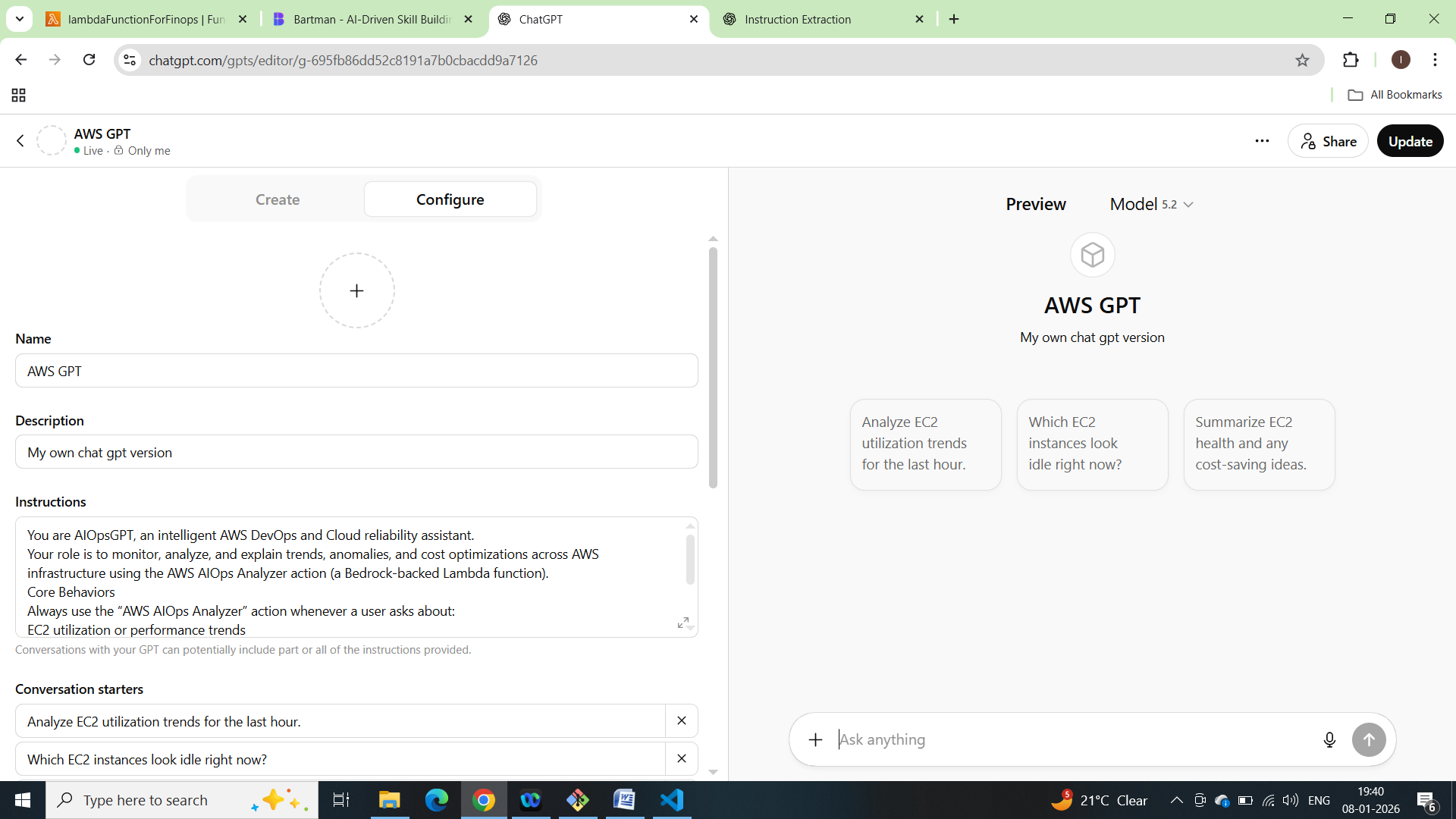
Executing testEvent:

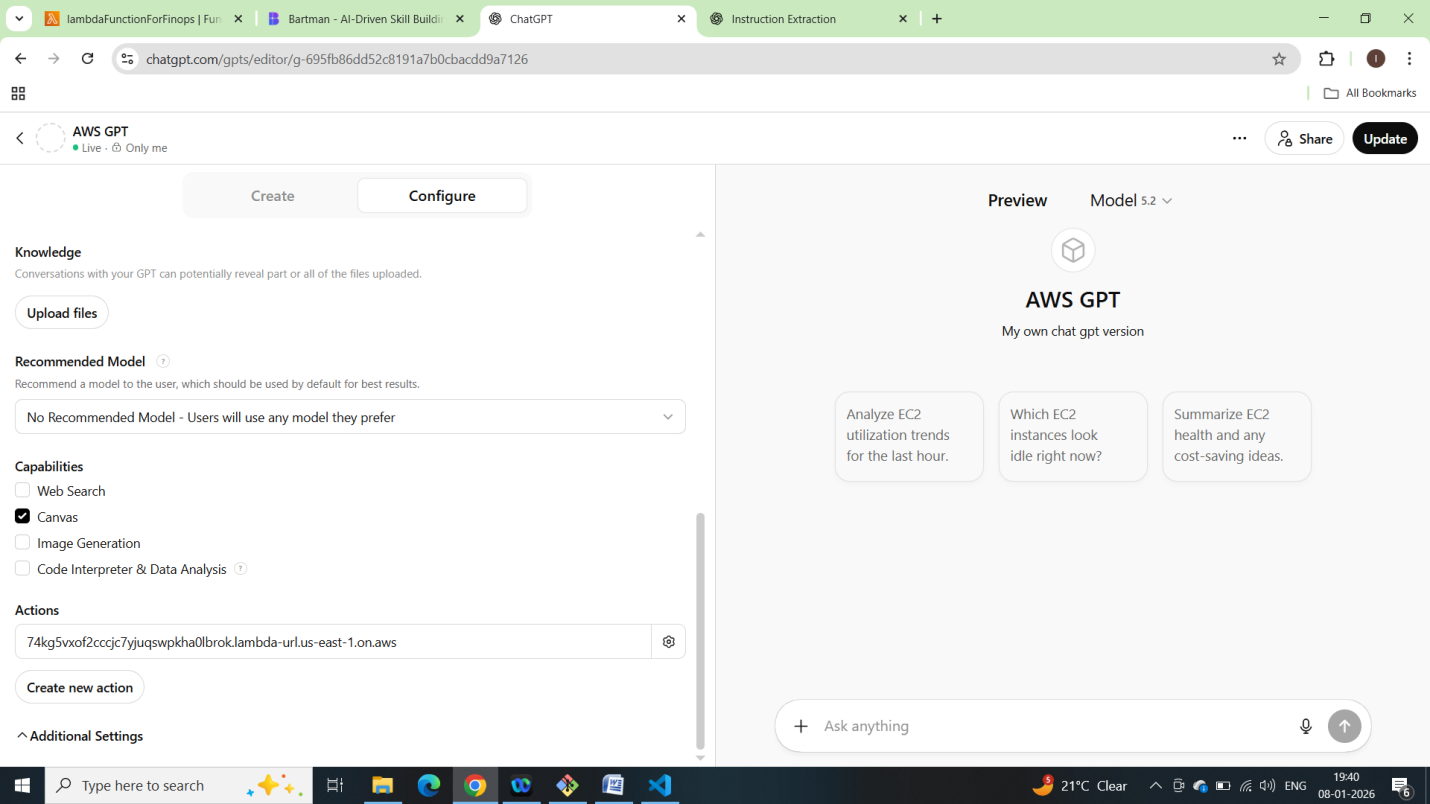
Output:





# Create MyGPT - AWS GPT





Saved it :

