Efficient Azure OpenAI API Usage with Function Calling, Batching, and Robust Retry Mechanisms

1. Objective

- Learn to use the OpenAI Python library with function calling to structure API interactions.
- Implement batching to handle multiple inputs efficiently.
- Apply rate limit handling using tenacity's decorators: `retry`, `wait_random_exponential`, and `stop_after_attempt`.
- Handle exceptions gracefully to ensure robustness during API calls.
- Complete a practical coding task within 120 minutes.

2. Problem Statement

- When using the Azure OpenAI API at scale, developers must manage complex workflows including multiple requests (batching), function calling to control output structure, and mitigating API rate limits via retries.
- Failing to handle these issues leads to degraded application performance or failures.
- This exercise guides you to build a Python module that processes a batch of inputs for a chosen topic using function calling, efficiently handles retries on rate limits, and manages exceptions properly.

3. Inputs / Shared Artifacts

- Choose one topic area for content generation (e.g., "Travel Itinerary Planning", "Recipe Summarization", or "Customer Support Email Drafting").
- No starter code provided — you will create:
  - Functions for batch processing with Azure OpenAI API calls
  - Retry wrappers using tenacity
  - Exception handling logic
- Environment setup:
- Python 3.x
  - `openai` library
  - `tenacity` library (`pip install tenacity`)
- Azure OpenAI Resource:
  - API key, endpoint URL, and deployment name (to be set as environment variables).

4. Expected Outcome

- A Python script/module that:
  - Accepts a list of input prompts (batch).
  - Uses Azure OpenAI API's function calling feature to request structured responses.
  - Retries on rate limit or transient errors with exponential backoff (max 5 attempts).
  - Properly logs or handles exceptions without crashing.
  - Demonstrates processing all batch inputs with final output collected and printed.
- Clear, modular, well-commented code.

5. Concepts Covered

- Azure OpenAI API usage with function calling
- Batching requests to optimize throughput
- Rate limit handling using tenacity decorators (`retry`, `wait_random_exponential`, `stop_after_attempt`)
- Exception handling and robustness
- Efficient Python asynchronous/synchronous programming for APIs (optional)
- Prompt and function schema design

6. Example: Step-by-Step Instructions with Code Snippet

```
from openai import AzureOpenAI
import time
from tenacity import retry, wait_random_exponential, stop_after_attempt,
retry_if_exception_type
from openai import RateLimitError, APIError
import os

os.environ["AZURE_OPENAI_ENDPOINT"] = ""
os.environ["AZURE_OPENAI_API_KEY"] = ""
os.environ["AZURE_DEPLOYMENT_NAME"]= "GPT-4o-mini"

# Step 1: Set your Azure OpenAI API key
client = AzureOpenAI(
    api_version="2024-07-01-preview",
    azure_endpoint=os.getenv("AZURE_OPENAI_ENDPOINT"),
    api_key=os.getenv("AZURE_OPENAI_API_KEY"),
)



# Step 2: Define your function schema for function calling
functions = [
    {
        "name": "generate_itinerary",
        "description": "Generate a travel itinerary for a given destination
and duration.",
```

```
        "parameters": {
            "type": "object",
            "properties": {
                "destination": {"type": "string", "description": "Travel
destination city or country"},
                "days": {"type": "integer", "description": "Number of days to
plan for"}
            },
            "required": ["destination", "days"],
        },
    }
]



# Step 3: Define a retry decorator for rate limits and transient errors
@retry(
    retry=retry_if_exception_type((RateLimitError, APIError)),
    wait=wait_random_exponential(min=1, max=10),
    stop=stop_after_attempt(5),
    reraise=True
)

def call_openai_function(prompt, destination, days):
    response = client.chat.completions.create(
        model=os.getenv("AZURE_DEPLOYMENT_NAME"),
        messages=[
            {"role": "user", "content": prompt}
        ],
        functions=functions,
        function_call={
            "name": "generate_itinerary",
            "arguments": f'{{"destination": "{destination}", "days":
{days}}}'
        }
    )
    return response



# Step 4: Batch processing function
def batch_process(inputs):
    results = []
    for input_data in inputs:
        try:
            prompt = input_data["prompt"]
            destination = input_data["destination"]
            days = input_data["days"]
            res = call_openai_function(prompt, destination, days)
            results.append(res)
            time.sleep(1)  # Optional: prevent hitting rate limits
aggressively
        except Exception as e:
            print(f"Error processing {destination}: {e}")
            results.append(None)
```

```
    return results

# Step 5: Example batch inputs

batch_inputs = [
    {"prompt": "Plan a travel itinerary.", "destination": "Paris", "days":
3},
    {"prompt": "Plan a travel itinerary.", "destination": "Tokyo", "days":
5},
    {"prompt": "Plan a travel itinerary.", "destination": "New York", "days":
4},
]

# Step 6: Run batch processing and display results

if __name__ == "__main__":
    outputs = batch_process(batch_inputs)
    for idx, output in enumerate(outputs):
        print(f"Result for {batch_inputs[idx]['destination']}:")
        if output:
            print(output)
        else:
            print("No result due to error or retry failure.")
        print("-" * 40)
```

7. Final Submission Checklist

- Submit your Python script/module with:
  - Proper function calling implementation
  - Batch processing capability
  - Retry logic with tenacity decorators
  - Exception handling code and error logging
- Code in .py file.
- Use dummy input list (auto input). Do not use manual input (input() built-in).
- Include sample inputs and outputs. (3 samples)
- Provide a brief README explaining:
  - How to run the script
  - Your design choices for batching and retries
  - Any challenges encountered and how they were solved