Mooli AI Chatbot: AI Usage Throttle

A Strategic Approach to Progressive Cost Management on AWS Bedrock

Prepared by: Tchokote N. Herve

Date: August 2025

The Problem: The High Cost of Uncontrolled AI

The High Cost of Uncontrolled AI

1. **Unpredictable Costs**: AI usage can spike unexpectedly, leading to massive bills.

2. **Poor User Experience**: Simple hard cutoffs at a usage limit frustrate users.

3. **The Goal**: Implement a system that gracefully manages usage without abrupt service interruption.

The Concept: Progressive Throttling

A system that **progressively slows down API response times** as a predefined usage threshold is approached.

This provides a “soft” signal to users while ensuring the budget isn’t exceeded.

Visual: Use a graphic to illustrate the different usage “zones”.

- **Green Zone** **(0-80%)**: Normal, fast performance.
- **Yellow Zone** **(80-95%)**: Introduce a minor, increasing delay.
- **Red Zone** **(95-100%)**: Significant delay with a warning message.
- **Exceeded (>100%)**: Restrict access to critical requests.

Implementation: At the Codebase Level of Mooli Chatbot

The Python code that handles the throttling logic.

- **Monitoring**: Using **LangChain's Callbacks** to track token usage for every LLM call. The on_llm_end callback is ideal as it provides input/output token counts.
- **State Management**: Store the current token count in a fast, shared data store like **Redis**. A simple global variable can be used for demonstration.
- **Throttle Logic**: We create a function (apply_progressive_throttle) that checks the current token count against the budget and applies a time.sleep() delay.

Python
```
# ai_chatbot/agent_tools.py
import time

DEMO_MAX_TOKENS = 5000
DEMO_CURRENT_TOKENS = 0

def apply_progressive_throttle():
    current_tokens = DEMO_CURRENT_TOKENS
    if current_tokens < DEMO_MAX_TOKENS * 0.8:
        return 0
    elif current_tokens >= DEMO_MAX_TOKENS * 0.8 and current_tokens <
DEMO_MAX_TOKENS * 0.95:
        # Exponentially increasing delay
        return 2 + (current_tokens / DEMO_MAX_TOKENS) * 10
    else:
        return 15 + (current_tokens / DEMO_MAX_TOKENS) * 20

def run_agent_task(user_input, file_path=None):
    # Apply delay before any LLM call
    delay = apply_progressive_throttle()
    if delay > 0:
        time.sleep(delay)
    # ... rest of the agent logic
```

Implementation: AWS Account & Infrastructure

Outline of the AWS services that support this solution.

- **Cost Management**: Use **AWS Budgets** to set budget thresholds and trigger alerts via SNS when usage crosses a certain percentage.
- **Cost Allocation**: Implement **AWS Cost Allocation Tags** to track costs per project or user. This allows for fine-grained budget control.
- **Storage**: Use a managed service like **Amazon ElastiCache for Redis** as the fast, central store for the token usage counter.
- **Networking (Advanced)**: For large-scale applications, API Gateway's throttling features can provide an extra layer of control.

Demonstration in the Mooli Chatbot

Description of a walk-through of the live demonstration.

1. **Setup**: The demonstration uses a simple token counter in the `agent_tools.py` file to simulate usage.

2. **Scenario 1 (Green Zone)**: A user sends a simple query. The chatbot responds instantly as normal.

3. **Scenario 2 (Yellow Zone)**: A manual script or a series of rapid queries is used to increase the token count. The user sends a new query and observes a noticeable, but not prohibitive, delay.

4. **Scenario 3 (Red Zone)**: The token count is pushed to near the limit. The user gets a slower response and a message like "High demand, response may be slow."

Technical Deep Dive & Code Snippets

I can provide code-level details for each component.

LangChain Callbacks: A custom `TokenUsageCallback` class that updates the Redis counter.

Django View Integration: Showing the updated `run_agent_task` function that calls `apply_progressive_throttle` before invoking the agent.

The Redis Usage: Showing a simple snippet of how to set and get the token counter from Redis.

Summary & Q&A

Key Takeaways:

- Proactive cost management is crucial for AI.
- Progressive throttling provides a better user experience than hard cutoffs.
- This solution combines software logic with scalable AWS services.

Final Statement: “This architecture allows for flexible, cost-aware AI services that scale with your business without surprises.”