(IJISE) 2023, Vol. No. 18, Jul-Dec **e-ISSN: 2454-6402, p-ISSN: 2454-812X**

Investigating the Influence of Cloud Computing and Edge Computing on Resource Usage for Mobile Devices Using Generative Artificial Intelligence APIs[1]

Tanvi S Hungund

Senior Manager, Dallas TX
California State University Fullerton

Received: 25 July 2023; Accepted: 23 September 2023; Published: 13 October 2023

ABSTRACT

The emergence of OpenAI's ChatGPT has sparked significant interest in generative AI within the business sphere. Generative AI APIs are now accessible on mobile devices, which face CPU, memory, and battery resource limitations. To conserve power, a combination of Edge and Cloud Computing can be employed. However, there's a trade-off with response times when implementing these services. Cloud computing tends to incur a higher response time penalty than edge computing, although payload limits and maximum response time constraints restrict the latter. Our testing indicates that Edge Computing strikes a good balance, saving CPU time while partially alleviating the response time penalty. Developers contemplating using Cloud and Edge Computing to save power on mobile devices should carefully evaluate their dataset sizes to determine the feasibility of employing an Edge Computing service.

INTRODUCTION

The proliferation of mobile subscriptions has shown consistent growth since 1993, reaching a staggering 8.6 billion in 2022, surpassing the global population [1]. Notably, in the United States alone, mobile devices accounted for 67.81% of website visits as of May 2023 [2]. Given these trends, it's foreseeable that mobile devices will constitute a significant portion of consumers utilizing internet-based application programming interfaces (APIs). However, mobile devices are constrained by limited resources such as battery life, memory, and smaller CPUs compared to desktop counterparts. Hence, understanding resource consumption based on different API utilization patterns on mobile devices becomes imperative.

One prominent API in focus is OpenAI's chat completions API, featuring ChatGPT, a form of generative artificial intelligence (AI) launched by OpenAI on November 30, 2022 [3]. ChatGPT facilitates text-based conversational interactions with AI models. OpenAI offers an interactive interface (https://chat.openai.com/) where users engage with ChatGPT by posing questions or statements and receiving responses based on its training data. For instance, querying ChatGPT about its model yields insightful responses, such as based on the GPT-3.5 architecture [4].

Developers can leverage OpenAI's Chat Completions API, which is similar to ChatGPT's functionality. This empowers the integration of OpenAI's models into diverse projects. Notably, the gut-turbo-3.5 model was selected for this project due to its generative text capabilities [5].

This project delves into utilizing the OpenAI API within a mobile application, scrutinizing resource utilization across three distinct API usage patterns. The mobile app orchestrates conversations with the OpenAI API, summarizing them akin to meeting notes. A comprehensive analysis of resource consumption aims to discern the

[1] *How to cite the article:* Hungund T.S.; Investigating the Influence of Cloud Computing and Edge Computing on Resource Usage for Mobile Devices Using Generative Artificial Intelligence APIs; *International Journal of Innovations in Scientific Engineering*, Jul-Dec 2023, Vol 18, 38-46

(IJISE) 2023, Vol. No. 18, Jul-Dec **e-ISSN: 2454-6402, p-ISSN: 2454-812X**

merits of Cloud and Edge Computing as alternatives to direct OpenAI API access. Amazon Web Services (AWS) Lambda is employed for Cloud Computing, while AWS Lambda@Edge is utilized for Edge Computing.

Prior research, such as the study by Linares-Vásquez, Bovata, et al., titled "Mining Energy-Greedy API Usage Patterns in Android Apps: An Empirical Study" [6], explores the energy implications of diverse API usage patterns on Android devices. Additionally, Qian & Anderson's work, "Reducing Mobile Device Energy Consumption with Computation Offloading" [7], investigates systems capable of offloading computations in Android applications to conserve energy.

IMPLEMENTATION

Utilization of the ChatGPT Completions API entailed sending a JSON object comprising a model property and a message collection. The model property specifies the model for processing the messages. Each Message contains two attributes: role and content. The role can assume one of four values: system, user, assistant, or function [8]. The "get-3.5-turbo" model was employed throughout all scenarios, with only the user and assistant roles utilized.

An illustrative snippet of a concise conversation input is presented in Figure 4. The completions API generated the content attributed to the "assistant" role in response to the preceding question. The content is truncated with ellipses for brevity, though the complete conversation would be included in the actual interaction.

```
{
    "model": "gpt-3.5-turbo",
    "messages": [{
                    "content": "What is a warm front?",
                    "role": "user"
            },
            {
                    "content": "A warm front is a ... snow.",
                    "role": "assistant"
            },
            {
                    "content": "What is a cold front?",
                    "role": "user"
            },
            {
                    "content": "A cold front is a ... thunderstorms.",
                    "role": "assistant"
            },
            {
                    "content": "Create meeting notes from this meeting",
                    "role": "user"
            }
    ]
}
```

Figure 1: Example JSON input for OpenAI Chat Completions API

```
{
    "id": "chatcmpl-7WFDcfOwhid5AqQxJn033qCUlI0Ud",
    "object": "chat.completion",
    "created": 1687919352,
    "model": "gpt-3.5-turbo-0613",
    "choices": [
        {
            "index": 0,
            "message": {
                "role": "assistant",
                "content": "Meeting Notes ... Any
                discrepancies or omissions are
                unintentional."
            },
            "finish_reason": "stop"
        }
    ],
    "usage": {
        "prompt_tokens": 186,
        "completion_tokens": 388,
        "total_tokens": 574
    }
}
```


Figure 2: Example JSON output from Chat Completions API

IMPLEMENTATION

Figure 2 depicts a condensed example of the output generated by the Chat Completions API, derived from the data outlined in Figure 1. Noteworthy properties for this endeavour encompass the "choices" collection and the message property of each item within. Each entry within "choices" (in this instance, singular) comprises a message object containing role and content attributes akin to the input object. The exhibited role pertains to the assistant, while the content serves as a condensed summary, resembling meeting notes from the preceding conversation. Notably, in Figure 1, the final Message prompts the API to generate meeting notes from the discussion.

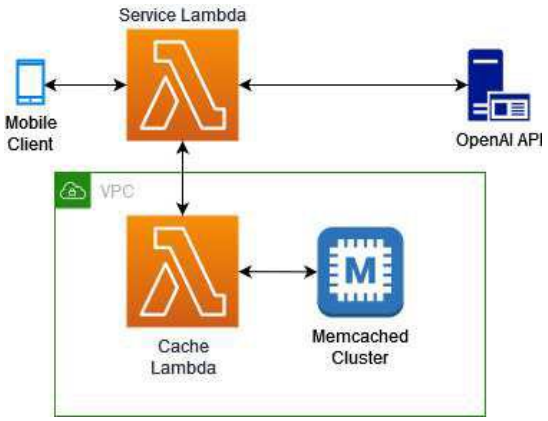

Figure 3: Cloud Computing Architecture

Cloud Computing services were facilitated through AWS Lambda. Lambda, an AWS serverless service [9], allows functions to be scripted in various languages, including Python, the language of choice for this project. The project necessitated both computational resources and transient storage. For the latter, AWS Elasticache with a Memcached server was opted for. Elasticache, a caching service compatible with Memcached or Redis, facilitates clustered cache servers, though a single-node configuration sufficed for this project.

Figure 3 presents an overview of the Cloud Computing Architecture. The "Service Lambda" constitutes the core component, facilitating communication with the mobile client, the OpenAI Chat Completions API, and the Cache Lambda. The "Cache Lambda" is responsible for transactions with the Memcached cluster.

A Virtual Private Cloud (VPC) container is integrated into the architecture. VPC, a VPN within the AWS Cloud, is secluded from other clients [10]. Initially, the Service and Cache Lambdas were amalgamated into a single Lambda within the VPC. However, it was discerned that to enable a Lambda function to access an external API like the OpenAI API, a NAT gateway is imperative, complicating the setup and inflating costs. AWS levies an hourly fee for the NAT gateway, with additional charges for data transmission. While a Lambda outside a VPC can access APIs, Elasticache services necessitate VPC placement. Hence, a Lambda outside a VPC was accorded permission to access a Lambda inside the VPC, establishing the chosen architecture.

OpenAI provides a Python library [11] utilized herein to interface with the OpenAI API. Code to communicate with the API was crafted using the OpenAI library, accompanying documentation, and guidance from Shameel Ahmed's tutorial on The Developer Space website [12].

```
def execCacheCmdAsync(payload):
    print(f"ExecCache, Payload:  {payload}")
    lambda_client = boto3.client('lambda')
    response = lambda_client.invoke(
        FunctionName = 'arn:aws:lambda:…'
        InvocationType = 'Event',
        Payload = payload
    )
```

Figure 4: Code calling Lamba in VPC from outside VPC

The Service Lambda utilizes the code excerpt in Figure 4 to communicate with the Cache Lambda. This code, drawing reference from a forum thread titled "Using boto to invoke lambda functions how do I do so asynchronously" [13] and AWS Boto3 documentation [14], enables asynchronous invocation of Lambda functions.

```
{
    "Version": "2022-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction",
                "lambda:InvokeAsync"
            ],
            "Resource": "arn:..."
        }
    ]
}
```

Figure 5: IAM Access Policy for Service Lambda to access Cache Lambda

IAM permissions were augmented through the AWS console to empower the invocation of the Lambda function within the VPC. Figure 4 illustrates the access policy, with the resource's actual ARN replacing the placeholder "arn:..." The permitted actions encompass synchronous (lambda: InvokeFunction) and asynchronous (lambda: InvokeAsync) invocation.

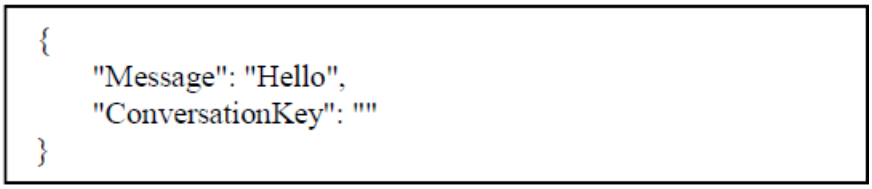

Figure 6: Example of JSON encoded data sent to start conversation.

The overarching process unfolds as follows:

- The client dispatches JSON-encoded data comprising the "Message" and "ConversationKey" fields. "Message" conveys the Message intended for the OpenAI API or the keyword "END", signalling conversation termination. "ConversationKey" denotes a unique conversation identifier assigned by the Service Lambda. A blank ConversationKey indicates the inception of a new conversation, prompting the Service Lambda to generate a new one upon response receipt. Figure 6 presents an example of JSON-encoded data forwarded to the Service Lambda to initiate a fresh conversation.

- The Service Lambda parses incoming data. If the ConversationKey is blank and the Message isn't "END", a new conversation key is assigned, triggering a request to the Cache Lambda to establish a new cache record.

- In scenarios where the Message isn't "END", an asynchronous request is sent to the Cache Lambda to append the Message to the conversation linked to the provided ConversationKey. Asynchronous invocation enables the Service Lambda to proceed without awaiting the Cache Lambda's response, thereby reducing client response time.

- Upon encountering "END" in the Message, a request is dispatched to the Cache Lambda to retrieve the conversation using the provided ConversationKey. Subsequently, a message attributed to the "user" role, with content instructing the creation of meeting notes from the discussion, is appended to the message list.

- The Service Lambda forwards the data (a single message or the entire conversation) to the OpenAI API, subsequently processing the response.

- If the response isn't meeting notes, an additional asynchronous call is placed to the Cache Lambda to incorporate the response into the conversation. This step is omitted in scenarios where the conversation concludes with meeting notes.

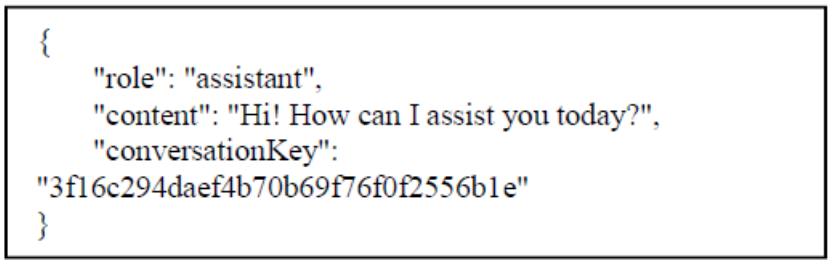

Figure 7: Example response from the Service Lambda

- Finally, the Service Lambda furnishes JSON-encoded data containing the OpenAI API's role and content alongside the conversation Key. Figure 7 presents an exemplary response from the Service Lambda.

The continuation of the process above persists until the conclusion of the conversation.

B. Edge Computing Implementation – AWS Lambda@Edge

The Edge Computing implementation mirrors the Cloud Computing setup with minor adjustments necessitated by platform constraints. For this purpose, AWS Lambda@Edge was selected. AWS Lambda@Edge enables the execution of Lambda functions in response to AWS CloudFront events [15], with AWS CloudFront serving as a Content Delivery Network (CDN) service offered by AWS [10].

AWS Lambda@Edge presents a pared-down version of AWS Lambda's functionalities, subject to various limitations detailed on Amazon's "Restrictions on edge functions" page [16]. Key limitations include:

- Support solely for Node.js and Python functions.

- A maximum zipped package size of 1 megabyte.

- Response time constraints capped at 30 seconds.

Creation of AWS Lambda@Edge functions involves converting a Lambda function into a Lambda@Edge function. However, including the OpenAI library and its dependencies inflated the Lambda package size beyond the 1MB limit. As the OpenAI library solely facilitated communication with the OpenAI API endpoint, dependency on it was eradicated. Instead, a Python function (depicted in Figure 8) was devised to manage communication with the OpenAI endpoint using the built-in Requests library.

```
def sendToOpenAI(messages, apikey, modelName, apiEndpoint):
      sendData = {"model":modelName, "messages": messages}
      sendheaders = {"Authorization": f"Bearer {apikey}"}
      response = requests.post(headers = sendheaders,
      json = sendData, url = apiEndpoint)
      resp = json.loads(response.content.decode(response.encoding))
      jresp = {"role":resp["choices"][0]["message"]["role"],
'content':resp["choices"][0]["message"]["content"]}
     return jresp
```

Figure 8: Python Code to Send/Receive Data from OpenAI API

Another Limitation and Its Mitigation

Another constraint impacted the asynchronous calls to the Cache Lambda. While the Cache Lambda required no modification, it couldn't be invoked asynchronously from the Lambda@Edge function. Unlike the AWS Lambda function, the Lambda@Edge function lacked access to the boto3. Client object, rendering asynchronous invocation unattainable.

Despite the alterations above, the process and data flow remains fundamentally akin to the AWS Lambda function. Furthermore, the output from the Lambda@Edge function mirrors that of the Lambda function. This congruence facilitates seamless interchangeability between the Lambda and Lambda@Edge implementations by adjusting the accessed endpoint URL. The Lambda@Edge function was forged by initially creating a Lambda function through the AWS browser-based console, subsequently transforming it into a Lambda@Edge function.

C. Direct Implementation and Mobile Application

The subsequent implementation phases were executed via a mobile application tailored for the Android platform, leveraging Android Studio 2022.2.1 Patch 1 [17]. Android Studio supports Java and Kotlin programming languages, with Kotlin selected for this venture.

The application encompasses two activities: the initial opening activity, which houses the main menu, and the subsequent activity, which realizes the chosen use case.

(IJISE) 2023, Vol. No. 18, Jul-Dec **e-ISSN: 2454-6402, p-ISSN: 2454-812X**

Upon clicking a main menu button, the second activity unfolds. This activity showcases an implementation activity featuring a user-sent message and the ensuing response received from the implemented source—in this case, Cloud Computing utilizing AWS Lambda. All three cases share a uniform interface. A parameter is relayed to the second activity upon selecting a use case, dictating the chosen data source. Two data provider classes were crafted for this initiative.

The Cloud Computing AWS Lambda and Edge Computing AWS Lambda@Edge functions yield output in identical formats. Consequently, they can leverage the same data provider class, differing solely in endpoint URLs. The second data provider class was tailored to interface directly with the OpenAI API. These providers conform to a unified class interface featuring a single method—sendMsg, mandating one parameter, "msgText." Both implementations leverage the Java.net HTTPURLConnection object to dispatch HTTP POST requests to the appropriate endpoint. Response times for these requests are tracked using timestamps, with cumulative response times monitored for testing purposes. Both implementations furnish the text of the Message returned from the OpenAI API.

Distinguishing the Cloud Computing/Edge implementation from the OpenAI implementation is the latter's requirement to execute tasks initially delegated to Cloud Computing and Edge Computing implementations. This encompasses direct communication with the OpenAI API, caching messages, and soliciting conversation summaries as meeting notes. Cached conversations are stored within the provider as a Mutable List of objects, each featuring role and content string properties.

Similar to the Lambda/Lambda@Edge implementations, the OpenAI message provider scrutinizes for the "END" keyword upon receiving a message. If absent, the Message is dispatched to OpenAI, processed, cached, and returned. However, upon encountering the "END" keyword, an item tagged as "user" with the content "Create meeting notes from this meeting" is appended to the conversation. Subsequently, the entire discussion is dispatched to the OpenAI endpoint.

The activity integrates a RecyclerView exhibiting the ongoing conversation alongside a text box enabling user message input. Below these elements are three buttons: "Send," "Benchmark," and "End Conversation." The "Send" button appends the Message from the text box to the RecyclerView, tags it as from the user, and transmits it to the message provider. Upon receiving a response from the provider, the corresponding Message is likewise appended to the RecyclerView and tagged as from the "bot" (OpenAI).

The "Benchmark" button facilitates application testing with a brief conversation, a process elaborated further in the ensuing Test Results section. Lastly, the "End Conversation" button executes analogous methods as the "Send" button, except it inputs "END" instead of the text box message. This keyword signifies conversation termination and a request to summarize the conversation as meeting notes. Moreover, this action turns off all controls on the screen, halting further conversation. Upon receipt of meeting notes from the provider, they are exhibited within the RecyclerView.

ANALYSIS OF RESULTS

A. CPU Time Analysis

The Edge Computing use case exhibited the lowest average CPU time when employing WiFi, approximately 1054 milliseconds, closely followed by Cloud Computing at around 1057 milliseconds on average. Utilizing cellular data, the Edge Computing use case marginally lagged behind the Cloud Computing scenario, with averages of about 1097ms versus 1077ms, respectively. Conversely, the direct communication with OpenAI's API registered the highest average CPU time across all cases. WiFi slightly outperformed cellular data for both Cloud Computing and Edge Computing scenarios, whereas cellular data slightly outperformed WiFi for the direct use case. Given that both Cloud and Edge Computing cases share identical code methods, with endpoint URLs being the sole disparity, their comparable performance was expected. The direct case, however, significantly underperformed, approximately 15% inferior to the combined average of the Cloud and Edge cases with WiFi, for instance. This discrepancy is unsurprising, considering the mobile device assumes processing tasks offloaded to Cloud and Edge resources in alternative use cases.

B. Response Time Analysis

The direct use case boasted the best average response time, owing to direct communication between the mobile device and OpenAI. Edge Computing trailed about 10% using WiFi and approximately 7% using cellular data. Cloud computing recorded the slowest average response time, about 33% slower using WiFi and roughly 19% slower using cellular data. Intriguingly, cellular data yielded faster response times than WiFi for Cloud and direct cases while proving slower for Edge scenarios. The direct use case's expedited response time is expected, circumventing intermediary services for direct interaction with the OpenAI API. Moreover, Edge Computing response time is anticipated to outpace Cloud Computing, considering the proximity of Edge devices to end users in most instances. Additionally, the connection from AWS infrastructure to OpenAI is expected to be comparable between Edge and Cloud Computing cases.

C. Device Memory Usage

Average memory usage peaked with the Edge Computing use case utilizing WiFi, slightly surpassing the direct use case and about 2.4% higher than Cloud Computing. Nonetheless, the Edge Computing case emerged as the lowest overall memory consumer, particularly with cellular data. Cloud Computing incurred marginally higher memory usage, while the direct use case also exhibited a slight increase. Furthermore, average memory consumption was higher for WiFi compared to cellular data. Despite Cloud and Edge Computing cases sharing identical code and data, differing only in URL endpoint usage, memory consumption was only partially comparable. Speculatively, memory allocation methods in the Android system might significantly influence these outcomes.

CONCLUSION

The project aimed to discern the advantages of mobile app utilization of Cloud Computing versus Edge Computing versus direct communication with the OpenAI Chat Completions API. While Cloud and Edge Computing yielded savings in CPU time, translating to potential battery power savings, a penalty was observed in response time. Edge Computing presents a viable middle ground, balancing CPU time savings and response time penalties. However, current AWS Lambda@Edge architecture constraints may limit Edge Computing's utility, particularly for projects involving larger datasets. Developers should carefully assess dataset sizes when considering Cloud and Edge Computing options for power savings on mobile devices, weighing response time tolerance against power savings advantages.

REFERENCES

1. P. Taylor, "Statistica: Number of movile (cellular) subscription sworldwide from 1993 to 2022," 1 June 2023. [Online]. Available: https://www.statista.com/statistics/262950/global-mobilesubscriptions-since-1993/. [Accessed 7 July 2032].
2. OpenAI, "Introducting ChatGPT," 30 November 2022. [Online]. Available: https://openai.com/blog/chatgpt.
3. M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta and D. Poshyvanyk, "Mining Energy-Greedy API Usage Patterns in Android Apps: An Empirical Study," in MSR '14, May 31-June 2, 2014, Hyderabad, 2014`.
4. H. Qian and D. Anderson, "Reducing Mobile Device Energy Consumption with Computation Offloading," in SNPD 2015, June 1-3 2015, Takamatsu, 2015.
5. OpenAI, "OpenAI API Reference," [Online]. Available: https://platform.openai.com/docs/api-reference. [Accessed 27 June 2023].
6. Amazon, "AWS Lambda Getting Started," [Online]. Available: https://aws.amazon.com/lambda/getting-started/. [Accessed 28 June 2023].
7. R. Fox and W. Hao, Internet Infrastructure: Networking, Web Services, and Cloud Computing, Boca Raton: CRC Press, 2018.
8. OpenAI, "OpenAI Libraries," [Online]. Available: https://platform.openai.com/docs/libraries. [Accessed 28 June 2023].
9. S. Ahmed, "The Developer Space - How to invoke OpenAI APIs from AWS Lambda functions," 5 February 2023. [Online]. Available: https://thedeveloperspace.com/how-to-invoke-openai-apis-from-awslambda-functions/. [Accessed 28 June 2023].

10. "Stackoverflow: Using boto to invoke functions how do I do so asynchronously?," 12 September 2016. [Online]. Available: https://stackoverflow.com/questions/39456309/using-boto-to-invokelambda-functions-how-do-i-do-so-asynchronously. [Accessed 28 June 2023].
11. Amazon Web Services, "Boto3 1.26.162 documentation: Lambda," [Online]. Available: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/lambda.html. [Accessed 28 June 2023].
12. Amazon Web Services, "Using AWS Lambda with CloudFront Lambda@Edge," [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html. [Accessed 30 June 2023].
13. Amazon Web Serivces, "Restrictions on edge functions," [Online]. Available:https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/edge-functions-restrictions.html. [Accessed 30 June 2023].
14. Google, "Android Studio," [Online]. Available: https://developer.android.com/studio. [Accessed 30 June 2023].