Batch API with OpenAI

- A Batch API allows clients to send **multiple requests** in a single HTTP call. This reduces overhead and network latency compared to making separate calls for every request.
- The Batch API returns completions within 24 hours for a **50% discount**. https://openai.com/api/pricing/
- The service is ideal for processing jobs that **don't require immediate** responses.
- The API typically expects the batch **request** to be formatted in a certain way, such as a JSON object containing an **array of individual requests**. Each request in the array might include details like the HTTP method, endpoint, headers, and body.
- Likewise, the **response** usually consists of an **array of responses**, each corresponding to an individual request. The sequence of responses should match the order of requests unless the API supports asynchronous processing with varied ids or status.
- While some uses of the OpenAI Platform require you to send synchronous requests, there are many cases where requests do not need an immediate response or rate limits prevent you from executing a large number of queries quickly.

Benefits of using BatchAI with OpenAI:

- **Scalability:** Process large datasets quickly by distributing tasks across multiple computing nodes.
- **Efficiency:** Optimize resource utilization by parallelizing computations.
- **Cost-Effectiveness:** Reduce processing time and associated costs by leveraging cloud-based computing power.

Model Availability

The Batch API can currently be used to execute queries against the following models. The Batch API supports text and vision inputs in the same format as the endpoints for these models:

- gpt-4o-mini
- gpt-4o
- gpt-4-turbo
- gpt-4
- gpt-3.5-turbo
- text-embedding-3-large
- text-embedding-3-small
- text-embedding-ada-002

The Batch API offers a straightforward set of endpoints that allow you

a) To collect a set of requests into a single file

b) Kick off a batch processing job to execute these requests

c) Query for the status of that batch while the underlying requests execute

d) Eventually retrieve the collected results when the batch is complete

Available Endpoints in batch:

a) /v1/chat/completions

b) /v1/embeddings

Example

Batches start with a .jsonl file where each line contains the details of an individual request to the API.

batch_input.json1

```
{"custom_id": "request-1", "method": "POST", "url": "/v1/chat/completions", "body": {"model": "gpt-4o-2024-05-13", "messages": [{"role": "system", "content": "You are a helpful assistant."},{"role": "user", "content": "Hello world!"}],"max_tokens": 1000}}
{"custom_id": "request-2", "method": "POST", "url": "/v1/chat/completions", "body": {"model": "gpt-4o-2024-05-13", "messages": [{"role": "system", "content": "You are an unhelpful assistant."},{"role": "user", "content": "Hello world!"}],"max_tokens": 1000}}
```

#Prepare and upload your batch file

```
batch_input_file = client.files.create(
  file=open("batch_input.json1", "rb"),
  purpose="batch"
)
print(batch_input_file)
```

#Creating the Batch

```
batch_input_file_id = batch_input_file.id
batch = client.batches.create(
  input_file_id=batch_input_file_id,
  endpoint="/v1/chat/completions",
  completion_window="24h",
  metadata={
    "description": "nightly eval job"
```

```
  }
)
print("Total: ", batch.request_counts.total)
print("Completed: ", batch.request_counts.completed)
print("Failed: ",batch.request_counts.failed)
```

#Listing existing batch Ids and status

```
batches = client.batches.list(limit=10)
for batch in batches.data:
  print("Id=", batch.id, " Status=", batch.status )
```

The status of a given Batch object can be any of the following:

STATUS	DESCRIPTION
validating	the input file is being validated before the batch can begin
failed	the input file has failed the validation process
in_progress	the input file was successfully validated and the batch is currently being run
finalizing	the batch has completed and the results are being prepared
completed	the batch has been completed and the results are ready
expired	the batch was not able to be completed within the 24-hour time window
cancelling	the batch is being cancelled (may take up to 10 minutes)
cancelled	the batch was cancelled

#Listing Batch tasks

```
batches = client.batches.list(limit=10)
for batch in batches.data:
  if batch.status == "in-progress":
    print(batch.id, " is in progress")
  #Checking the Status of a Batch
  if batch.status == "completed":
    #Retrieving the batch results
    file_response = client.files.content(batch.output_file_id)
    file_content = file_response.text
    lines = file_content.splitlines()
    for line in lines:
      line_json = json.loads(line)
```

```
        print(line_json["response"]['body']['choices'][0]['message']['content'])
```

All in one Code

```
import openai
import json
from util import getOpenAIClient

client = getOpenAIClient()

def upload_file_and_create_batch():
    # Upload batch tasks file
    with open("batch_input.json1",  "rb") as f:
        batch_input_file = client.files.create(
            file=f,
            purpose="batch"
        )
    print("File uploaded with ID: ", batch_input_file.id)

    # Create and execute a batch from an uploaded  file of requests
    batch = client.batches.create(
        input_file_id=batch_input_file.id,
        endpoint="/v1/chat/completions",
        completion_window="24h",
        metadata={
            "description": "nightly eval job"
        }
    )
    print("Batch created with ID: ", batch.id)
    return batch

def print_batch_ids():
    batches = client.batches.list(limit=10)
    for batch in batches: #["data"]:
        print(f"Batch Id= {batch.id} Status= {batch.status}")
```

```
def print_batch_details(batch_id):
    batch = client.batches.retrieve(batch_id)
    print("Status=", batch.status)
    print("Total Tasks: ", batch.request_counts.total)
    print("Completed Tasks: ", batch.request_counts.completed)
    print("Failed Tasks: ", batch.request_counts.failed)

    if batch.status == "completed":
        file_response = client.files.content(batch.output_file_id)
        file_content = file_response.text
        lines = file_content.splitlines()
        for line in lines:
            if line.strip():
                line_json = json.loads(line)
                print("Response: ", line_json["response"])
                print(line_json["response"]["body"]["choices"][0]["message"]["content"])
                print("--------------------")

def get_batch_id_from_user():
    batch_id = input("Enter batch ID: ")
    return batch_id

def delete_batch(batch_id):
    client.batches.delete(batch_id)
    print("Batch deleted")

def main():
    print("Select an option:")
    print("1. Upload File and Create Batch")
    print("2. Get Batch Ids")
    print("3. Print Batch Details")

    choice = input("Enter your choice: ")
    if choice == '1':
        upload_file_and_create_batch()
```

```
    elif choice == '2':
        print_batch_ids()
    elif choice == '3':
        batch_to_print = get_batch_id_from_user()
        print_batch_details(batch_to_print)
    else:
        print("Invalid choice")

# Run the main function
main()
```

Rate Limits

Batch API rate limits are separate from existing per-model rate limits. The Batch API has two new types of rate limits:

1. **Per-batch limits:** A single batch may include up to 50,000 requests, and a batch input file can be up to 200 MB in size. Note that /v1/embeddings batches are also restricted to a maximum of 50,000 embedding inputs across all requests in the batch.
2. **Enqueued prompt tokens per model**: Each model has a maximum number of enqueued prompt tokens allowed for batch processing. You can find these limits on the Platform Settings page.

There are no limits for output tokens or number of submitted requests for the Batch API today. Because Batch API rate limits are a new, separate pool, using the Batch API will not consume tokens from your standard per-model rate limits, thereby offering you a convenient way to increase the number of requests and processed tokens you can use when querying our API.

Batch Expiration

- Batches that do not complete in time eventually move to an **expired** state; unfinished requests within that batch are **cancelled**, and any responses to completed requests are made available via the batch's output file. You will be charged for tokens consumed from any completed requests.
- Expired requests will be written to your error file with the message as shown below. You can use the **custom_id** to retrieve the request data for expired requests.

```
{"id": "batch_req_123", "custom_id": "request-3", "response": null, "error": {"code": "batch_expired",
"message": "This request could not be executed before the completion window expired."}}
{"id": "batch_req_123", "custom_id": "request-7", "response": null, "error": {"code": "batch_expired",
"message": "This request could not be executed before the completion window expired."}}
```