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**.
* 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**

1. To collect a set of requests into a single file,
2. Kick off a batch processing job to execute these requests,
3. Query for the status of that batch while the underlying requests execute,
4. Eventually retrieve the collected results when the batch is complete

**Available Endpoints in batch:**

1. /v1/chat/completions
2. /v1/embeddings

**Batch API:**

<https://platform.openai.com/docs/api-reference/batch>

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 |

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-3.5-turbo-0125", "messages": [{"role": "system", "content": "You are a helpful assistant."},{"role": "user", "content": "What is OpenAI"}],"max\_tokens": 1000}}

{"custom\_id": "request-2", "method": "POST", "url": "/v1/chat/completions", "body": {"model": "gpt-3.5-turbo-0125", "messages": [{"role": "system", "content": "You are an helpful assistant."},{"role": "user", "content": "What is Python"}],"max\_tokens": 1000}}

**main.js**

import fs from 'fs';

import readline from 'readline';

import { generateToken } from './util.js';

import { OpenAI  } from 'openai';

await generateToken()

const client = new OpenAI();

async function uploadFileAndCreateBatch() {

    //Upload batch tasks file

    const batchInputFile = await client.files.create({

        file: fs.createReadStream('batch\_input.json1'),

        purpose: 'batch'

    });

    console.log("File uploaded with ID: ", batchInputFile.id);

    //Creates and executes a batch from an uploaded file of requests

    const batch = await client.batches.create({

        input\_file\_id: batchInputFile.id,

        endpoint: "/v1/chat/completions",

        completion\_window: "24h",

        metadata: {

            description: "nightly eval job"

        }

    });

    console.log("Batch created with ID: ", batch.id);

    return batch;

}

async function printBatchIds() {

    const batches = await client.batches.list({ limit: 10 });

    for (let batch of batches.data) {

        console.log("Batch Id= " + batch.id + " Status=" + batch.status);

    }

}

async function printBatchDetails(batchId) {

    const batch = await client.batches.retrieve(batchId);

    console.log(" Status=", batch.status)

    console.log("Total Tasks: ", batch.request\_counts.total);

    console.log("Completed Tasks: ", batch.request\_counts.completed);

    console.log("Failed: Tasks", batch.request\_counts.failed);

    if (batch.status === "completed") {

        const fileResponse = await client.files.content(batch.output\_file\_id);

        const fileContent = await fileResponse.text();

        const lines = fileContent.split('\n');

        for (let line of lines) {

            if (line.trim()) {

                const lineJson = JSON.parse(line);

                console.log("Response: ", lineJson.response);

                console.log(lineJson.response.body.choices[0].message.content);

                console.log("--------------------");

            }

        }

    }

}

async function getBatchIdFromUser() {

    return new Promise((resolve) => {

        const rl = readline.createInterface({

            input: process.stdin,

            output: process.stdout

        });

        rl.question("Enter batch ID: ", (batchId) => {

            rl.close();

            resolve(batchId);

        });

    });

}

async function main() {

    console.log("Select an option:");

    console.log("1. Upload File and Create Batch");

    console.log("2. Get Batch Ids")

    console.log("3. Print Batch Details");

    const rl = readline.createInterface({

        input: process.stdin,

        output: process.stdout

    });

    rl.question("Enter your choice: ", async (choice) => {

        switch (choice) {

            case '1':

                await uploadFileAndCreateBatch();

                break;

            case '2':

                await printBatchIds();

                break;

            case '3':

                const batchToPrint = await getBatchIdFromUser();

                await printBatchDetails(batchToPrint);

                break;

            default:

                console.log("Invalid choice");

        }

        rl.close();

    });

}

await 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 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."}}