"""

API REQUEST PARALLEL PROCESSOR

Using the OpenAI API to process lots of text quickly takes some care.

If you trickle in a million API requests one by one, they'll take days to complete.

If you flood a million API requests in parallel, they'll exceed the rate limits and fail with errors.

To maximize throughput, parallel requests need to be throttled to stay under rate limits.

This script parallelizes requests to the OpenAI API while throttling to stay under rate limits.

Features:

- Streams requests from file, to avoid running out of memory for giant jobs

- Makes requests concurrently, to maximize throughput

- Throttles request and token usage, to stay under rate limits

- Retries failed requests up to {max\_attempts} times, to avoid missing data

- Logs errors, to diagnose problems with requests

Example command to call script:

```

python examples/api\_request\_parallel\_processor.py \

--requests\_filepath examples/data/example\_requests\_to\_parallel\_process.jsonl \

--save\_filepath examples/data/example\_requests\_to\_parallel\_process\_results.jsonl \

--request\_url https://api.openai.com/v1/embeddings \

--max\_requests\_per\_minute 1500 \

--max\_tokens\_per\_minute 6250000 \

--token\_encoding\_name cl100k\_base \

--max\_attempts 5 \

--logging\_level 20

```

Inputs:

- requests\_filepath : str

- path to the file containing the requests to be processed

- file should be a jsonl file, where each line is a json object with API parameters and an optional metadata field

- e.g., {"model": "text-embedding-3-small", "input": "embed me", "metadata": {"row\_id": 1}}

- as with all jsonl files, take care that newlines in the content are properly escaped (json.dumps does this automatically)

- an example file is provided at examples/data/example\_requests\_to\_parallel\_process.jsonl

- the code to generate the example file is appended to the bottom of this script

- save\_filepath : str, optional

- path to the file where the results will be saved

- file will be a jsonl file, where each line is an array with the original request plus the API response

- e.g., [{"model": "text-embedding-3-small", "input": "embed me"}, {...}]

- if omitted, results will be saved to {requests\_filename}\_results.jsonl

- request\_url : str, optional

- URL of the API endpoint to call

- if omitted, will default to "https://api.openai.com/v1/embeddings"

- api\_key : str, optional

- API key to use

- if omitted, the script will attempt to read it from an environment variable {os.getenv("OPENAI\_API\_KEY")}

- max\_requests\_per\_minute : float, optional

- target number of requests to make per minute (will make less if limited by tokens)

- leave headroom by setting this to 50% or 75% of your limit

- if requests are limiting you, try batching multiple embeddings or completions into one request

- if omitted, will default to 1,500

- max\_tokens\_per\_minute : float, optional

- target number of tokens to use per minute (will use less if limited by requests)

- leave headroom by setting this to 50% or 75% of your limit

- if omitted, will default to 125,000

- token\_encoding\_name : str, optional

- name of the token encoding used, as defined in the `tiktoken` package

- if omitted, will default to "cl100k\_base" (used by `text-embedding-3-small`)

- max\_attempts : int, optional

- number of times to retry a failed request before giving up

- if omitted, will default to 5

- logging\_level : int, optional

- level of logging to use; higher numbers will log fewer messages

- 40 = ERROR; will log only when requests fail after all retries

- 30 = WARNING; will log when requests his rate limits or other errors

- 20 = INFO; will log when requests start and the status at finish

- 10 = DEBUG; will log various things as the loop runs to see when they occur

- if omitted, will default to 20 (INFO).

The script is structured as follows:

- Imports

- Define main()

- Initialize things

- In main loop:

- Get next request if one is not already waiting for capacity

- Update available token & request capacity

- If enough capacity available, call API

- The loop pauses if a rate limit error is hit

- The loop breaks when no tasks remain

- Define dataclasses

- StatusTracker (stores script metadata counters; only one instance is created)

- APIRequest (stores API inputs, outputs, metadata; one method to call API)

- Define functions

- api\_endpoint\_from\_url (extracts API endpoint from request URL)

- append\_to\_jsonl (writes to results file)

- num\_tokens\_consumed\_from\_request (bigger function to infer token usage from request)

- task\_id\_generator\_function (yields 0, 1, 2, ...)

- Run main()

"""

# imports

import aiohttp # for making API calls concurrently

import argparse # for running script from command line

import asyncio # for running API calls concurrently

import json # for saving results to a jsonl file

import logging # for logging rate limit warnings and other messages

import os # for reading API key

import re # for matching endpoint from request URL

import tiktoken # for counting tokens

import time # for sleeping after rate limit is hit

from dataclasses import (

dataclass,

field,

) # for storing API inputs, outputs, and metadata

async def process\_api\_requests\_from\_file(

requests\_filepath: str,

save\_filepath: str,

request\_url: str,

api\_key: str,

max\_requests\_per\_minute: float,

max\_tokens\_per\_minute: float,

token\_encoding\_name: str,

max\_attempts: int,

logging\_level: int,

):

"""Processes API requests in parallel, throttling to stay under rate limits."""

# constants

seconds\_to\_pause\_after\_rate\_limit\_error = 15

seconds\_to\_sleep\_each\_loop = (

0.001 # 1 ms limits max throughput to 1,000 requests per second

)

# initialize logging

logging.basicConfig(level=logging\_level)

logging.debug(f"Logging initialized at level {logging\_level}")

# infer API endpoint and construct request header

api\_endpoint = api\_endpoint\_from\_url(request\_url)

request\_header = {"Authorization": f"Bearer {api\_key}"}

# use api-key header for Azure deployments

if "/deployments" in request\_url:

request\_header = {"api-key": f"{api\_key}"}

# initialize trackers

queue\_of\_requests\_to\_retry = asyncio.Queue()

task\_id\_generator = (

task\_id\_generator\_function()

) # generates integer IDs of 0, 1, 2, ...

status\_tracker = (

StatusTracker()

) # single instance to track a collection of variables

next\_request = None # variable to hold the next request to call

# initialize available capacity counts

available\_request\_capacity = max\_requests\_per\_minute

available\_token\_capacity = max\_tokens\_per\_minute

last\_update\_time = time.time()

# initialize flags

file\_not\_finished = True # after file is empty, we'll skip reading it

logging.debug(f"Initialization complete.")

# initialize file reading

with open(requests\_filepath) as file:

# `requests` will provide requests one at a time

requests = file.\_\_iter\_\_()

logging.debug(f"File opened. Entering main loop")

async with aiohttp.ClientSession() as session: # Initialize ClientSession here

while True:

# get next request (if one is not already waiting for capacity)

if next\_request is None:

if not queue\_of\_requests\_to\_retry.empty():

next\_request = queue\_of\_requests\_to\_retry.get\_nowait()

logging.debug(

f"Retrying request {next\_request.task\_id}: {next\_request}"

)

elif file\_not\_finished:

try:

# get new request

request\_json = json.loads(next(requests))

next\_request = APIRequest(

task\_id=next(task\_id\_generator),

request\_json=request\_json,

token\_consumption=num\_tokens\_consumed\_from\_request(

request\_json, api\_endpoint, token\_encoding\_name

),

attempts\_left=max\_attempts,

metadata=request\_json.pop("metadata", None),

)

status\_tracker.num\_tasks\_started += 1

status\_tracker.num\_tasks\_in\_progress += 1

logging.debug(

f"Reading request {next\_request.task\_id}: {next\_request}"

)

except StopIteration:

# if file runs out, set flag to stop reading it

logging.debug("Read file exhausted")

file\_not\_finished = False

# update available capacity

current\_time = time.time()

seconds\_since\_update = current\_time - last\_update\_time

available\_request\_capacity = min(

available\_request\_capacity

+ max\_requests\_per\_minute \* seconds\_since\_update / 60.0,

max\_requests\_per\_minute,

)

available\_token\_capacity = min(

available\_token\_capacity

+ max\_tokens\_per\_minute \* seconds\_since\_update / 60.0,

max\_tokens\_per\_minute,

)

last\_update\_time = current\_time

# if enough capacity available, call API

if next\_request:

next\_request\_tokens = next\_request.token\_consumption

if (

available\_request\_capacity >= 1

and available\_token\_capacity >= next\_request\_tokens

):

# update counters

available\_request\_capacity -= 1

available\_token\_capacity -= next\_request\_tokens

next\_request.attempts\_left -= 1

# call API

asyncio.create\_task(

next\_request.call\_api(

session=session,

request\_url=request\_url,

request\_header=request\_header,

retry\_queue=queue\_of\_requests\_to\_retry,

save\_filepath=save\_filepath,

status\_tracker=status\_tracker,

)

)

next\_request = None # reset next\_request to empty

# if all tasks are finished, break

if status\_tracker.num\_tasks\_in\_progress == 0:

break

# main loop sleeps briefly so concurrent tasks can run

await asyncio.sleep(seconds\_to\_sleep\_each\_loop)

# if a rate limit error was hit recently, pause to cool down

seconds\_since\_rate\_limit\_error = (

time.time() - status\_tracker.time\_of\_last\_rate\_limit\_error

)

if (

seconds\_since\_rate\_limit\_error

< seconds\_to\_pause\_after\_rate\_limit\_error

):

remaining\_seconds\_to\_pause = (

seconds\_to\_pause\_after\_rate\_limit\_error

- seconds\_since\_rate\_limit\_error

)

await asyncio.sleep(remaining\_seconds\_to\_pause)

# ^e.g., if pause is 15 seconds and final limit was hit 5 seconds ago

logging.warn(

f"Pausing to cool down until {time.ctime(status\_tracker.time\_of\_last\_rate\_limit\_error + seconds\_to\_pause\_after\_rate\_limit\_error)}"

)

# after finishing, log final status

logging.info(

f"""Parallel processing complete. Results saved to {save\_filepath}"""

)

if status\_tracker.num\_tasks\_failed > 0:

logging.warning(

f"{status\_tracker.num\_tasks\_failed} / {status\_tracker.num\_tasks\_started} requests failed. Errors logged to {save\_filepath}."

)

if status\_tracker.num\_rate\_limit\_errors > 0:

logging.warning(

f"{status\_tracker.num\_rate\_limit\_errors} rate limit errors received. Consider running at a lower rate."

)

# dataclasses

@dataclass

class StatusTracker:

"""Stores metadata about the script's progress. Only one instance is created."""

num\_tasks\_started: int = 0

num\_tasks\_in\_progress: int = 0 # script ends when this reaches 0

num\_tasks\_succeeded: int = 0

num\_tasks\_failed: int = 0

num\_rate\_limit\_errors: int = 0

num\_api\_errors: int = 0 # excluding rate limit errors, counted above

num\_other\_errors: int = 0

time\_of\_last\_rate\_limit\_error: int = 0 # used to cool off after hitting rate limits

@dataclass

class APIRequest:

"""Stores an API request's inputs, outputs, and other metadata. Contains a method to make an API call."""

task\_id: int

request\_json: dict

token\_consumption: int

attempts\_left: int

metadata: dict

result: list = field(default\_factory=list)

async def call\_api(

self,

session: aiohttp.ClientSession,

request\_url: str,

request\_header: dict,

retry\_queue: asyncio.Queue,

save\_filepath: str,

status\_tracker: StatusTracker,

):

"""Calls the OpenAI API and saves results."""

logging.info(f"Starting request #{self.task\_id}")

error = None

try:

async with session.post(

url=request\_url, headers=request\_header, json=self.request\_json

) as response:

response = await response.json()

if "error" in response:

logging.warning(

f"Request {self.task\_id} failed with error {response['error']}"

)

status\_tracker.num\_api\_errors += 1

error = response

if "rate limit" in response["error"].get("message", "").lower():

status\_tracker.time\_of\_last\_rate\_limit\_error = time.time()

status\_tracker.num\_rate\_limit\_errors += 1

status\_tracker.num\_api\_errors -= (

1 # rate limit errors are counted separately

)

except (

Exception

) as e: # catching naked exceptions is bad practice, but in this case we'll log & save them

logging.warning(f"Request {self.task\_id} failed with Exception {e}")

status\_tracker.num\_other\_errors += 1

error = e

if error:

self.result.append(error)

if self.attempts\_left:

retry\_queue.put\_nowait(self)

else:

logging.error(

f"Request {self.request\_json} failed after all attempts. Saving errors: {self.result}"

)

data = (

[self.request\_json, [str(e) for e in self.result], self.metadata]

if self.metadata

else [self.request\_json, [str(e) for e in self.result]]

)

append\_to\_jsonl(data, save\_filepath)

status\_tracker.num\_tasks\_in\_progress -= 1

status\_tracker.num\_tasks\_failed += 1

else:

data = (

[self.request\_json, response, self.metadata]

if self.metadata

else [self.request\_json, response]

)

append\_to\_jsonl(data, save\_filepath)

status\_tracker.num\_tasks\_in\_progress -= 1

status\_tracker.num\_tasks\_succeeded += 1

logging.debug(f"Request {self.task\_id} saved to {save\_filepath}")

# functions

def api\_endpoint\_from\_url(request\_url):

"""Extract the API endpoint from the request URL."""

match = re.search("^https://[^/]+/v\\d+/(.+)$", request\_url)

if match is None:

# for Azure OpenAI deployment urls

match = re.search(

r"^https://[^/]+/openai/deployments/[^/]+/(.+?)(\?|$)", request\_url

)

return match[1]

def append\_to\_jsonl(data, filename: str) -> None:

"""Append a json payload to the end of a jsonl file."""

json\_string = json.dumps(data)

with open(filename, "a") as f:

f.write(json\_string + "\n")

def num\_tokens\_consumed\_from\_request(

request\_json: dict,

api\_endpoint: str,

token\_encoding\_name: str,

):

"""Count the number of tokens in the request. Only supports completion and embedding requests."""

encoding = tiktoken.get\_encoding(token\_encoding\_name)

# if completions request, tokens = prompt + n \* max\_tokens

if api\_endpoint.endswith("completions"):

max\_tokens = request\_json.get("max\_tokens", 15)

n = request\_json.get("n", 1)

completion\_tokens = n \* max\_tokens

# chat completions

if api\_endpoint.startswith("chat/"):

num\_tokens = 0

for message in request\_json["messages"]:

num\_tokens += 4 # every message follows <im\_start>{role/name}\n{content}<im\_end>\n

for key, value in message.items():

num\_tokens += len(encoding.encode(value))

if key == "name": # if there's a name, the role is omitted

num\_tokens -= 1 # role is always required and always 1 token

num\_tokens += 2 # every reply is primed with <im\_start>assistant

return num\_tokens + completion\_tokens

# normal completions

else:

prompt = request\_json["prompt"]

if isinstance(prompt, str): # single prompt

prompt\_tokens = len(encoding.encode(prompt))

num\_tokens = prompt\_tokens + completion\_tokens

return num\_tokens

elif isinstance(prompt, list): # multiple prompts

prompt\_tokens = sum([len(encoding.encode(p)) for p in prompt])

num\_tokens = prompt\_tokens + completion\_tokens \* len(prompt)

return num\_tokens

else:

raise TypeError(

'Expecting either string or list of strings for "prompt" field in completion request'

)

# if embeddings request, tokens = input tokens

elif api\_endpoint == "embeddings":

input = request\_json["input"]

if isinstance(input, str): # single input

num\_tokens = len(encoding.encode(input))

return num\_tokens

elif isinstance(input, list): # multiple inputs

num\_tokens = sum([len(encoding.encode(i)) for i in input])

return num\_tokens

else:

raise TypeError(

'Expecting either string or list of strings for "inputs" field in embedding request'

)

# more logic needed to support other API calls (e.g., edits, inserts, DALL-E)

else:

raise NotImplementedError(

f'API endpoint "{api\_endpoint}" not implemented in this script'

)

def task\_id\_generator\_function():

"""Generate integers 0, 1, 2, and so on."""

task\_id = 0

while True:

yield task\_id

task\_id += 1

# run script

if \_\_name\_\_ == "\_\_main\_\_":

# parse command line arguments

parser = argparse.ArgumentParser()

parser.add\_argument("--requests\_filepath")

parser.add\_argument("--save\_filepath", default=None)

parser.add\_argument("--request\_url", default="https://api.openai.com/v1/embeddings")

parser.add\_argument("--api\_key", default=os.getenv("OPENAI\_API\_KEY"))

parser.add\_argument("--max\_requests\_per\_minute", type=int, default=3\_000 \* 0.5)

parser.add\_argument("--max\_tokens\_per\_minute", type=int, default=250\_000 \* 0.5)

parser.add\_argument("--token\_encoding\_name", default="cl100k\_base")

parser.add\_argument("--max\_attempts", type=int, default=5)

parser.add\_argument("--logging\_level", default=logging.INFO)

args = parser.parse\_args()

if args.save\_filepath is None:

args.save\_filepath = args.requests\_filepath.replace(".jsonl", "\_results.jsonl")

# run script

asyncio.run(

process\_api\_requests\_from\_file(

requests\_filepath=args.requests\_filepath,

save\_filepath=args.save\_filepath,

request\_url=args.request\_url,

api\_key=args.api\_key,

max\_requests\_per\_minute=float(args.max\_requests\_per\_minute),

max\_tokens\_per\_minute=float(args.max\_tokens\_per\_minute),

token\_encoding\_name=args.token\_encoding\_name,

max\_attempts=int(args.max\_attempts),

logging\_level=int(args.logging\_level),

)

)

"""

APPENDIX

The example requests file at openai-cookbook/examples/data/example\_requests\_to\_parallel\_process.jsonl contains 10,000 requests to text-embedding-3-small.

It was generated with the following code:

```python

import json

filename = "data/example\_requests\_to\_parallel\_process.jsonl"

n\_requests = 10\_000

jobs = [{"model": "text-embedding-3-small", "input": str(x) + "\n"} for x in range(n\_requests)]

with open(filename, "w") as f:

for job in jobs:

json\_string = json.dumps(job)

f.write(json\_string + "\n")

```

As with all jsonl files, take care that newlines in the content are properly escaped (json.dumps does this automatically).

"""