Rate limits

Rate limits are restrictions that our API imposes on the number of times a user or client can access our services within a specified period of time.

Why do we have rate limits?

Rate limits are a common practice for APIs, and they're put in place for a few different reasons:

- **They help protect against abuse or misuse of the API.** For example, a malicious actor could flood the API with requests in an attempt to overload it or cause disruptions in service. By setting rate limits, OpenAI can prevent this kind of activity.
- **Rate limits help ensure that everyone has fair access to the API.** If one person or organization makes an excessive number of requests, it could bog down the API for everyone else. By throttling the number of requests that a single user can make, OpenAI ensures that the most number of people have an opportunity to use the API without experiencing slowdowns.
- **Rate limits can help OpenAI manage the aggregate load on its infrastructure.** If requests to the API increase dramatically, it could tax the servers and cause performance issues. By setting rate limits, OpenAI can help maintain a smooth and consistent experience for all users.

> Please work through this document in its entirety to better understand how OpenAI's rate limit system works. We include code examples and possible solutions to handle common issues. We also include details around how your rate limits are automatically increased in the usage tiers section below.

How do these rate limits work?

Rate limits are measured in five ways: **RPM** (requests per minute), **RPD** (requests per day), **TPM** (tokens per minute), **TPD** (tokens per day), and **IPM** (images per minute). Rate limits can be hit across any of the options depending on what occurs first. For example, you might send 20 requests with only 100 tokens to the ChatCompletions endpoint and that would fill your limit (if your RPM was 20), even if you did not send 150k tokens (if your TPM limit was 150k) within those 20 requests.

Batch API queue limits are calculated based on the total number of input tokens queued for a given model. Tokens from pending batch jobs are counted against your queue limit. Once a batch job is completed, its tokens are no longer counted against that model's limit.

Other important things worth noting:

- Rate limits are defined at the organization level and at the project level, not user level.

- Rate limits vary by the model being used.
- Limits are also placed on the total amount an organization can spend on the API each month. These are also known as "usage limits".

Usage tiers

You can view the rate and usage limits for your organization under the limits section of your account settings. As your usage of the OpenAI API and your spend on our API goes up, we automatically graduate you to the next usage tier. This usually results in an increase in rate limits across most models.

TIER	QUALIFICATION	USAGE LIMITS
Free	User must be in an allowed geography	$100 / month
Tier 1	$5 paid	$100 / month
Tier 2	$50 paid and 7+ days since first successful payment	$500 / month
Tier 3	$100 paid and 7+ days since first successful payment	$1,000 / month
Tier 4	$250 paid and 14+ days since first successful payment	$5,000 / month
Tier 5	$1,000 paid and 30+ days since first successful payment	$15,000 / month

Select a tier below to view a high-level summary of rate limits per model.

Free | Tier 1 | Tier 2 | Tier 3 | Tier 4 | Tier 5

Free tier rate limits

This is a high level summary and there are per-model exceptions to these limits (e.g. some legacy models or models with larger context windows have different rate limits). To view the exact rate limits per model for your account, visit the limits section of your account settings.

MODEL	RPM	RPD	TPM	BATCH QUEUE LIMIT
`gpt-3.5-turbo`	3	200	40,000	200,000
`text-embedding-3-large`	3,000	200	1,000,000	3,000,000
`text-embedding-3-small`	3,000	200	1,000,000	3,000,000
`text-embedding-ada-002`	3,000	200	1,000,000	3,000,000

MODEL	RPM	RPD	TPM	BATCH QUEUE LIMIT
whisper-1	3	200	-	-
tts-1	3	200	-	-
dall-e-2	5 img/min	-	-	-
dall-e-3	1 img/min	-	-	-

Rate limits in headers

In addition to seeing your rate limit on your account page, you can also view important information about your rate limits such as the remaining requests, tokens, and other metadata in the headers of the HTTP response.

You can expect to see the following header fields:

FIELD	SAMPLE VALUE	DESCRIPTION
x-ratelimit-limit-requests	60	The maximum number of requests that are permitted before exhausting the rate limit.
x-ratelimit-limit-tokens	150000	The maximum number of tokens that are permitted before exhausting the rate limit.
x-ratelimit-remaining-requests	59	The remaining number of requests that are permitted before exhausting the rate limit.
x-ratelimit-remaining-tokens	149984	The remaining number of tokens that are permitted before exhausting the rate limit.
x-ratelimit-reset-requests	1s	The time until the rate limit (based on requests) resets to its initial state.
x-ratelimit-reset-tokens	6m0s	The time until the rate limit (based on tokens) resets to its initial state.

Error Mitigation

What are some steps I can take to mitigate this?

The OpenAI Cookbook has a Python notebook that explains how to avoid rate limit errors, as well an example Python script for staying under rate limits while batch processing API requests.

You should also exercise caution when providing programmatic access, bulk processing features, and automated social media posting - consider only enabling these for trusted customers.

To protect against automated and high-volume misuse, set a usage limit for individual users within a specified time frame (daily, weekly, or monthly). Consider implementing a hard cap or a manual review process for users who exceed the limit.

Retrying with exponential backoff

One easy way to avoid rate limit errors is to automatically retry requests with a random exponential backoff. Retrying with exponential backoff means performing a short sleep when a rate limit error is hit, then retrying the unsuccessful request. If the request is still unsuccessful, the sleep length is increased and the process is repeated. This continues until the request is successful or until a maximum number of retries is reached. This approach has many benefits:

- Automatic retries means you can recover from rate limit errors without crashes or missing data
- Exponential backoff means that your first retries can be tried quickly, while still benefiting from longer delays if your first few retries fail
- Adding random jitter to the delay helps retries from all hitting at the same time.

Note that unsuccessful requests contribute to your per-minute limit, so continuously resending a request won't work.

Below are a few example solutions **for Python** that use exponential backoff.

Example 1: Using the Tenacity library

Tenacity is an Apache 2.0 licensed general-purpose retrying library, written in Python, to simplify the task of adding retry behavior to just about anything. To add exponential backoff to your requests, you can use the `tenacity.retry` decorator. The below example uses the `tenacity.wait_random_exponential` function to add random exponential backoff to a request.

Using the Tenacity library — python

```python
from openai import OpenAI
client = OpenAI()

from tenacity import (
    retry,
    stop_after_attempt,
    wait_random_exponential,
```

```python
) # for exponential backoff

@retry(wait=wait_random_exponential(min=1, max=60), stop=stop_after_attempt(6))
def completion_with_backoff(**kwargs):
    return client.completions.create(**kwargs)

completion_with_backoff(model="gpt-3.5-turbo-instruct", prompt="Once upon a time,")
```

Note that the Tenacity library is a third-party tool, and OpenAI makes no guarantees about its reliability or security.

Example 2: Using the backoff library

Another python library that provides function decorators for backoff and retry is backoff:

Using the Tenacity library — python

```python
import backoff
import openai
from openai import OpenAI
client = OpenAI()

@backoff.on_exception(backoff.expo, openai.RateLimitError)
def completions_with_backoff(**kwargs):
    return client.completions.create(**kwargs)

completions_with_backoff(model="gpt-3.5-turbo-instruct", prompt="Once upon a time,")
```

Like Tenacity, the backoff library is a third-party tool, and OpenAI makes no guarantees about its reliability or security.

Example 3: Manual backoff implementation

If you don't want to use third-party libraries, you can implement your own backoff logic following this example:

Using manual backoff implementation — python

```python
# imports
import random
```

```python
import time

import openai
from openai import OpenAI
client = OpenAI()

# define a retry decorator
def retry_with_exponential_backoff(
    func,
    initial_delay: float = 1,
    exponential_base: float = 2,
    jitter: bool = True,
    max_retries: int = 10,
    errors: tuple = (openai.RateLimitError,),
):
    """Retry a function with exponential backoff."""

    def wrapper(*args, **kwargs):
        # Initialize variables
        num_retries = 0
        delay = initial_delay

        # Loop until a successful response or max_retries is hit or an exception is rai
        while True:
            try:
                return func(*args, **kwargs)

            # Retry on specific errors
            except errors as e:
                # Increment retries
                num_retries += 1

                # Check if max retries has been reached
                if num_retries > max_retries:
                    raise Exception(
                        f"Maximum number of retries ({max_retries}) exceeded."
                    )

                # Increment the delay
                delay *= exponential_base * (1 + jitter * random.random())

                # Sleep for the delay
                time.sleep(delay)

            # Raise exceptions for any errors not specified
            except Exception as e:
                raise e

```

```
    return wrapper

@retry_with_exponential_backoff
def completions_with_backoff(**kwargs):
    return client.completions.create(**kwargs)
```

Again, OpenAI makes no guarantees on the security or efficiency of this solution but it can be a good starting place for your own solution.

Reduce the `max_tokens` to match the size of your completions

Your rate limit is calculated as the maximum of `max_tokens` and the estimated number of tokens based on the character count of your request. Try to set the `max_tokens` value as close to your expected response size as possible.

Batching requests

If your use case does not require immediate responses, you can use the Batch API to more easily submit and execute large collections of requests without impacting your synchronous request rate limits.

For use cases that *do* requires synchronous respones, the OpenAI API has separate limits for **requests per minute** and **tokens per minute**.

If you're hitting the limit on requests per minute but have available capacity on tokens per minute, you can increase your throughput by batching multiple tasks into each request. This will allow you to process more tokens per minute, especially with our smaller models.

Sending in a batch of prompts works exactly the same as a normal API call, except you pass in a list of strings to the prompt parameter instead of a single string.

Example without batching

No batching python

```python
from openai import OpenAI
client = OpenAI()

num_stories = 10
prompt = "Once upon a time,"
```

```
6
7  # serial example, with one story completion per request
8  for _ in range(num_stories):
9      response = client.completions.create(
10         model="curie",
11         prompt=prompt,
12         max_tokens=20,
13     )
14     # print story
15     print(prompt + response.choices[0].text)
```

Example with batching

Batching python

```
1  from openai import OpenAI
2  client = OpenAI()
3
4  num_stories = 10
5  prompts = ["Once upon a time,"] * num_stories
6
7  # batched example, with 10 story completions per request
8  response = client.completions.create(
9      model="curie",
10     prompt=prompts,
11     max_tokens=20,
12 )
13
14 # match completions to prompts by index
15 stories = [""] * len(prompts)
16 for choice in response.choices:
17     stories[choice.index] = prompts[choice.index] + choice.text
18
19 # print stories
20 for story in stories:
21     print(story)
```

Warning: the response object may not return completions in the order of the prompts, so always remember to match responses back to prompts using the index field.