# Cerebras Python API library

[![PyPI
version](https://img.shields.io/pypi/v/cerebras_cloud_sdk.svg)](https://pypi.org/project/cerebras_cloud_sdk/)

The Cerebras Python library provides convenient access to the Cerebras REST API from any Python 3.8+

application. The library includes type definitions for all request params and response fields,

and offers both synchronous and asynchronous clients powered by
[httpx](https://github.com/encode/httpx).

It is generated with [Stainless](https://www.stainlessapi.com/).

## About Cerebras

At Cerebras, we've developed the world's largest and fastest AI processor, the Wafer-Scale Engine-3 (WSE-3). The Cerebras CS-3 system, powered by the WSE-3, represents a new class of AI supercomputer that sets the standard for generative AI training and inference with unparalleled performance and scalability.

With Cerebras as your inference provider, you can:

- Achieve unprecedented speed for AI inference workloads

- Build commercially with high throughput

- Effortlessly scale your AI workloads with our seamless clustering technology

Our CS-3 systems can be quickly and easily clustered to create the largest AI supercomputers in the world, making it simple to place and run the largest models. Leading corporations, research institutions, and governments are already using Cerebras solutions to develop proprietary models and train popular open-source models.

Want to experience the power of Cerebras? Check out our [website](https://cerebras.net) for more resources and explore options for accessing our technology through the Cerebras Cloud or on-premise deployments!

> [!NOTE]

> This SDK has a mechanism that sends a few requests to `/v1/tcp_warming` upon construction to reduce the TTFT. If this behaviour is not desired, set `warm_tcp_connection=False` in the constructor.

>

> If you are repeatedly reconstructing the SDK instance it will lead to poor performance. It is recommended that you construct the SDK once and reuse the instance if possible.

## Documentation

The REST API documentation can be found on inference-docs.cerebras.ai. The full API of this library can be found in [api.md](api.md).

## Installation

```

pip install cerebras_cloud_sdk

```

## API Key

Get an API Key from cloud.cerebras.ai and add it to your environment variables:

```

export CEREBRAS_API_KEY="your-api-key-here"

```

## Usage

The full API of this library can be found in [api.md](api.md).

### Chat Completion

<!-- RUN TEST: ChatStandard -->

```python

import os
```

```
from cerebras.cloud.sdk import Cerebras

client = Cerebras(
    api_key=os.environ.get("CEREBRAS_API_KEY"),  # This is the default and can be omitted
)

chat_completion = client.chat.completions.create(
    messages=[
        {
            "role": "user",
            "content": "Why is fast inference important?",
        }
    ],
    model="llama3.1-8b",
)

print(chat_completion)
```

### Text Completion
<!-- RUN TEST: TextStandard -->
```python
import os
from cerebras.cloud.sdk import Cerebras

client = Cerebras(
    api_key=os.environ.get("CEREBRAS_API_KEY"),  # This is the default and can be omitted
)

completion = client.completions.create(
    prompt="It was a dark and stormy ",
    max_tokens=100,
    model="llama3.1-8b",
)

print(completion)
```

While you can provide an `api_key` keyword argument,
we recommend using [python-dotenv](https://pypi.org/project/python-dotenv/)
to add `CEREBRAS_API_KEY="My API Key"` to your `.env` file
so that your API Key is not stored in source control.

## Async usage

Simply import `AsyncCerebras` instead of `Cerebras` and use `await` with each API call:

<!-- RUN TEST: ChatAsync -->
```python
import os
import asyncio
from cerebras.cloud.sdk import AsyncCerebras

client = AsyncCerebras(
    api_key=os.environ.get("CEREBRAS_API_KEY"),  # This is the default and can be omitted
)

async def main() -> None:
    chat_completion = await client.chat.completions.create(
        messages=[
            {
                "role": "user",
                "content": "Why is fast inference important?",
            }
        ],
        model="llama3.1-8b",
    )
    print(chat_completion)

asyncio.run(main())
```

Functionality between the synchronous and asynchronous clients is otherwise identical.

## Streaming responses

We provide support for streaming responses using Server Side Events (SSE).

Note that when streaming, `usage` and `time_info` will be information will only be included in the final chunk.

### Chat Completion

<!-- RUN TEST: ChatStreaming -->

```python
import os
from cerebras.cloud.sdk import Cerebras

client = Cerebras(
    # This is the default and can be omitted
    api_key=os.environ.get("CEREBRAS_API_KEY"),
)

stream = client.chat.completions.create(
    messages=[
        {
            "role": "user",
            "content": "Why is fast inference important?",
        }
    ],
    model="llama3.1-8b",
    stream=True,
)

for chunk in stream:
    print(chunk.choices[0].delta.content or "", end="")
```

The async client uses the exact same interface.

<!-- RUN TEST: ChatAsyncStreaming -->
```python
import os
import asyncio
from cerebras.cloud.sdk import AsyncCerebras

client = AsyncCerebras(
    # This is the default and can be omitted
    api_key=os.environ.get("CEREBRAS_API_KEY"),
)
```

async def main() -> None:
    stream = await client.chat.completions.create(
        messages=[
            {
                "role": "user",
                "content": "Why is fast inference important?",
            }
        ],
        model="llama3.1-8b",
        stream=True,
)
    async for chunk in stream:
        print(chunk.choices[0].delta.content or "", end="")

asyncio.run(main())
~~~

Text Completion

<!-- RUN TEST: TextStreaming -->

```python
import os
from cerebras.cloud.sdk import Cerebras

client = Cerebras(
    # This is the default and can be omitted
    api_key=os.environ.get("CEREBRAS_API_KEY"),
)

stream = client.completions.create(
    prompt="It was a dark and stormy ",
~~~

```
    max_tokens=100,
    model="llama3.1-8b",
    stream=True,
)

for chunk in stream:
    print(chunk.choices[0].text or "", end="")
```

## Using types

Nested request parameters are [TypedDicts](https://docs.python.org/3/library/typing.html#typing.TypedDict). Responses are [Pydantic models](https://docs.pydantic.dev) which also provide helper methods for things like:

- Serializing back into JSON, `model.to_json()`
- Converting to a dictionary, `model.to_dict()`

Typed requests and responses provide autocomplete and documentation within your editor. If you would like to see type errors in VS Code to help catch bugs earlier, set `python.analysis.typeCheckingMode` to `basic`.

## Handling errors

When the library is unable to connect to the API (for example, due to network connection problems or a timeout), a subclass of `cerebras.cloud.sdk.APIConnectionError` is raised.

When the API returns a non-success status code (that is, 4xx or 5xx

response), a subclass of `cerebras.cloud.sdk.APIStatusError` is raised, containing `status_code` and `response` properties.

All errors inherit from `cerebras.cloud.sdk.APIError`.

<!-- RUN TEST: Error -->

```python
import cerebras.cloud.sdk
from cerebras.cloud.sdk import Cerebras

client = Cerebras()

try:
    client.chat.completions.create(
        messages=[
            {
                "role": "user",
                "content": "This should cause an error!",
            }
        ],
        model="some-model-that-doesnt-exist",
    )
except cerebras.cloud.sdk.APIConnectionError as e:
    print("The server could not be reached")
    print(e.__cause__)  # an underlying Exception, likely raised within httpx.
except cerebras.cloud.sdk.RateLimitError as e:
    print("A 429 status code was received; we should back off a bit.")
except cerebras.cloud.sdk.APIStatusError as e:
    print("Another non-200-range status code was received")
    print(e.status_code)
    print(e.response)
```

Error codes are as followed:

Status Code	Error Type
400	`BadRequestError`
401	`AuthenticationError`
403	`PermissionDeniedError`
404	`NotFoundError`
422	`UnprocessableEntityError`
429	`RateLimitError`
>=500	`InternalServerError`
N/A	`APIConnectionError`

### Retries

Certain errors are automatically retried 2 times by default, with a short exponential backoff.

Connection errors (for example, due to a network connectivity problem), 408 Request Timeout, 409 Conflict,

429 Rate Limit, and >=500 Internal errors are all retried by default.

You can use the `max_retries` option to configure or disable retry settings:

<!-- RUN TEST: Retries -->

```python
from cerebras.cloud.sdk import Cerebras


# Configure the default for all requests:
client = Cerebras(
    # default is 2
    max_retries=0,
)
```

# Or, configure per-request:
client.with_options(max_retries=5).chat.completions.create(
    messages=[
        {
            "role": "user",
            "content": "Why is fast inference important?",
        }
    ],
    model="llama3.1-8b",
)
~~~

Timeouts

By default requests time out after 1 minute. You can configure this with a `timeout` option,
which accepts a float or an [`httpx.Timeout`](https://www.python-httpx.org/advanced/#fine-tuning-the-configuration) object:

<!-- RUN TEST: Timeout -->
```python
from cerebras.cloud.sdk import Cerebras
import httpx

# Configure the default for all requests:
client = Cerebras(
    # 20 seconds (default is 1 minute)
    timeout=20.0,
)
~~~

```
# More granular control:
client = Cerebras(
    timeout=httpx.Timeout(60.0, read=5.0, write=10.0, connect=2.0),
)

# Override per-request:
client.with_options(timeout=5.0).chat.completions.create(
    messages=[
        {
            "role": "user",
            "content": "Why is fast inference important?",
        }
    ],
    model="llama3.1-8b",
)
```

On timeout, an `APITimeoutError` is thrown.

Note that requests that time out are [retried twice by default](#retries).

## Advanced

### Logging

We use the standard library [`logging`](https://docs.python.org/3/library/logging.html) module.

You can enable logging by setting the environment variable `CEREBRAS_LOG` to `info`.

```shell
```

```
$ export CEREBRAS_LOG=info
```

Or to `debug` for more verbose logging.

### How to tell whether `None` means `null` or missing

In an API response, a field may be explicitly `null`, or missing entirely; in either case, its value is `None` in this library. You can differentiate the two cases with `.model_fields_set`:

```py
if response.my_field is None:
  if 'my_field' not in response.model_fields_set:
    print('Got json like {}, without a "my_field" key present at all.')
  else:
    print('Got json like {"my_field": null}.')
```

### Accessing raw response data (e.g. headers)

The "raw" Response object can be accessed by prefixing `.with_raw_response.` to any HTTP method call, e.g.,

<!-- RUN TEST: Advanced -->

```py
from cerebras.cloud.sdk import Cerebras

client = Cerebras()
response = client.chat.completions.with_raw_response.create(
    messages=[{
        "role": "user",
```

        "content": "Why is fast inference important?",
    }],
    model="llama3.1-8b",
)
print(response.headers.get('X-My-Header'))

completion = response.parse() # get the object that `chat.completions.create()` would have returned
print(completion)
~~~

These methods return an [`APIResponse`](https://github.com/Cerebras/cerebras-cloud-sdk-python/tree/main/src/cerebras/cloud/sdk/_response.py) object.

The async client returns an [`AsyncAPIResponse`](https://github.com/Cerebras/cerebras-cloud-sdk-python/tree/main/src/cerebras/cloud/sdk/_response.py) with the same structure, the only difference being `await`able methods for reading the response content.

<!--

`.with_streaming_response`

The above interface eagerly reads the full response body when you make the request, which may not always be what you want.

To stream the response body, use `.with_streaming_response` instead, which requires a context manager and only reads the response body once you call `.read()`, `.text()`, `.json()`, `.iter_bytes()`, `.iter_text()`, `.iter_lines()` or `.parse()`. In the async client, these are async methods.

```python
with client.chat.completions.with_streaming_response.create(
    messages=[
        {
            "role": "user",
~~~

```
            "content": "Why is fast inference important?",
        }
    ],
    model="llama3.1-8b",
) as response:
    print(response.headers.get("X-My-Header"))

    for line in response.iter_lines():
        print(line)
```

The context manager is required so that the response will reliably be closed.

-->

### Making custom/undocumented requests

This library is typed for convenient access to the documented API.

If you need to access undocumented endpoints, params, or response properties, the library can still be used.

#### Undocumented endpoints

To make requests to undocumented endpoints, you can make requests using `client.get`, `client.post`, and other

http verbs. Options on the client will be respected (such as retries) will be respected when making this

request.

```py
import httpx
```

```
response = client.post(
    "/foo",
    cast_to=httpx.Response,
    body={"my_param": True},
)

print(response.headers.get("x-foo"))
```

#### Undocumented request params

If you want to explicitly send an extra param, you can do so with the `extra_query`, `extra_body`, and `extra_headers` request
options.

#### Undocumented response properties

To access undocumented response properties, you can access the extra fields like `response.unknown_prop`. You
can also get all the extra fields on the Pydantic model as a dict with
[`response.model_extra`](https://docs.pydantic.dev/latest/api/base_model/#pydantic.BaseModel.model_extra).

### Configuring the HTTP client

You can directly override the [httpx client](https://www.python-httpx.org/api/#client) to customize it for your use case, including:

- Support for [proxies](https://www.python-httpx.org/advanced/proxies/)
- Custom [transports](https://www.python-httpx.org/advanced/transports/)
- Additional [advanced](https://www.python-httpx.org/advanced/clients/) functionality

```python
import httpx
from cerebras.cloud.sdk import Cerebras, DefaultHttpxClient

client = Cerebras(
    # Or use the `CEREBRAS_BASE_URL` env var
    base_url="http://my.test.server.example.com:8083",
    http_client=DefaultHttpxClient(
        proxy="http://my.test.proxy.example.com",
        transport=httpx.HTTPTransport(local_address="0.0.0.0"),
    ),
)
```

You can also customize the client on a per-request basis by using `with_options()`:

```python
client.with_options(http_client=DefaultHttpxClient(...))
```

### Managing HTTP resources

By default the library closes underlying HTTP connections whenever the client is [garbage collected](https://docs.python.org/3/reference/datamodel.html#object.__del__). You can manually close the client using the `.close()` method if desired, or with a context manager that closes when exiting.

```py
from cerebras.cloud.sdk import Cerebras

with Cerebras() as client:
```

  # make requests here

  ...

# HTTP client is now closed

~~~

Versioning

This package generally follows [SemVer](https://semver.org/spec/v2.0.0.html) conventions, though certain backwards-incompatible changes may be released as minor versions:

1. Changes that only affect static types, without breaking runtime behavior.

2. Changes to library internals which are technically public but not intended or documented for external use. _*(Please open a GitHub issue to let us know if you are relying on such internals)*_.

3. Changes that we do not expect to impact the vast majority of users in practice.

We take backwards-compatibility seriously and work hard to ensure you can rely on a smooth upgrade experience.

We are keen for your feedback; please open an [issue](https://www.github.com/Cerebras/cerebras-cloud-sdk-python/issues) with questions, bugs, or suggestions.

Determining the installed version

If you've upgraded to the latest version but aren't seeing any new features you were expecting then your python environment is likely still using an older version.

You can determine the version that is being used at runtime with:

```py

import cerebras.cloud.sdk

print(cerebras.cloud.sdk.__version__)
~~~

```

Requirements

Python 3.8 or higher.

Contributing

See [the contributing documentation](./CONTRIBUTING.md).
```