runwayml 3.8.0

`pip install runwayml`

Latest version

Released: Jul 31, 2025

The official Python library for the runwayml API

Verified details

These details have been verified by PyPI

Maintainers

runway

Unverified details

*These details have **not** been verified by PyPI*

Project links

- Homepage
- Repository

Meta

- **License:** Apache Software License (Apache-2.0)
- **Author:** RunwayML
- **Requires:** Python >=3.8
- **Provides-Extra:** `aiohttp`

Classifiers

Intended Audience
- Developers

License
- OSI Approved :: Apache Software License

Operating System
- MacOS
- Microsoft :: Windows
- OS Independent
- POSIX
- POSIX :: Linux

Programming Language
- Python :: 3.8
- Python :: 3.9
- Python :: 3.10
- Python :: 3.11
- Python :: 3.12
- Python :: 3.13

Topic
- Software Development :: Libraries :: Python Modules

Typing
- Typed

Project description

RunwayML Python API library

pypi (stable) v3.8.0

The RunwayML Python library provides convenient access to the RunwayML 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.

It is generated with Stainless.

Documentation

The REST API documentation can be found on dev.runwayml.com. The full API of this library can be found in api.md.

Installation

```
# install from PyPI
pip install runwayml
```

Usage

The full API of this library can be found in api.md.

```python
import os
from runwayml import RunwayML

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

image_to_video = client.image_to_video.create(
    model="gen4_turbo",
    prompt_image="https://example.com/assets/bunny.jpg",
    ratio="1280:720",
    prompt_text="The bunny is eating a carrot",
)
print(image_to_video.id)
```

While you can provide a `api_key` keyword argument, we recommend using python-dotenv to add `RUNWAYML_API_SECRET="My API Key"` to your `.env` file so that your API Key is not stored in source control.

Async usage

Simply import `AsyncRunwayML` instead of `RunwayML` and use `await` with each API call:

```python
import os
import asyncio
from runwayml import AsyncRunwayML

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


async def main() -> None:
    image_to_video = await client.image_to_video.create(
        model="gen4_turbo",
        prompt_image="https://example.com/assets/bunny.jpg",
        ratio="1280:720",
        prompt_text="The bunny is eating a carrot",
    )
    print(image_to_video.id)


asyncio.run(main())
```

Functionality between the synchronous and asynchronous clients is otherwise identical.

With aiohttp

By default, the async client uses `httpx` for HTTP requests. However, for improved concurrency performance you may also use `aiohttp` as the HTTP backend.

You can enable this by installing `aiohttp`:

```
# install from PyPI
pip install runwayml[aiohttp]
```

Then you can enable it by instantiating the client with `http_client=DefaultAioHttpClient()`:

```python
import asyncio
from runwayml import DefaultAioHttpClient
from runwayml import AsyncRunwayML


async def main() -> None:
    async with AsyncRunwayML(
        api_key="My API Key",
        http_client=DefaultAioHttpClient(),
    ) as client:
        image_to_video = await client.image_to_video.create(
            model="gen4_turbo",
            prompt_image="https://example.com/assets/bunny.jpg",
            ratio="1280:720",
            prompt_text="The bunny is eating a carrot",
        )
        print(image_to_video.id)


asyncio.run(main())
```

Using types

Nested request parameters are TypedDicts. Responses are Pydantic models 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`.

Nested params

Nested parameters are dictionaries, typed using `TypedDict`, for example:

```python
from runwayml import RunwayML

client = RunwayML()

text_to_image = client.text_to_image.create(
    model="gen4_image",
    prompt_text="promptText",
    ratio="1920:1080",
)
print(text_to_image.id)
```

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 `runwayml.APIConnectionError` is raised.

When the API returns a non-success status code (that is, 4xx or 5xx response), a subclass of `runwayml.APIStatusError` is raised, containing `status_code` and `response` properties.

All errors inherit from `runwayml.APIError`.

```python
import runwayml
from runwayml import RunwayML

client = RunwayML()

try:
    client.image_to_video.create(
        model="gen4_turbo",
        prompt_image="https://example.com/assets/bunny.jpg",
        ratio="1280:720",
        prompt_text="The bunny is eating a carrot",
    )
except runwayml.APIConnectionError as e:
    print("The server could not be reached")
    print(e.__cause__)  # an underlying Exception, likely raised within httpx.
except runwayml.RateLimitError as e:
    print("A 429 status code was received; we should back off a bit.")
except runwayml.APIStatusError as e:
    print("Another non-200-range status code was received")
    print(e.status_code)
    print(e.response)
```

Error codes are as follows:

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:

```python
from runwayml import RunwayML

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

# Or, configure per-request:
client.with_options(max_retries=5).image_to_video.create(
    model="gen4_turbo",
    prompt_image="https://example.com/assets/bunny.jpg",
    ratio="1280:720",
    prompt_text="The bunny is eating a carrot",
)
```

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` object:

```python
from runwayml import RunwayML

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

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

# Override per-request:
client.with_options(timeout=5.0).image_to_video.create(
    model="gen4_turbo",
    prompt_image="https://example.com/assets/bunny.jpg",
    ratio="1280:720",
    prompt_text="The bunny is eating a carrot",
)
```

On timeout, an `APITimeoutError` is thrown.

Note that requests that time out are retried twice by default.

Advanced

Logging

We use the standard library `logging` module.

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

```
$ export RUNWAYML_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`:

```python
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.,

```python
from runwayml import RunwayML

client = RunwayML()
response = client.image_to_video.with_raw_response.create(
    model="gen4_turbo",
    prompt_image="https://example.com/assets/bunny.jpg",
    ratio="1280:720",
    prompt_text="The bunny is eating a carrot",
)
print(response.headers.get('X-My-Header'))

image_to_video = response.parse()  # get the object that `image_to_video.create()` would have
print(image_to_video.id)
```

These methods return an `APIResponse` object.

The async client returns an `AsyncAPIResponse` 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.image_to_video.with_streaming_response.create(
    model="gen4_turbo",
    prompt_image="https://example.com/assets/bunny.jpg",
    ratio="1280:720",
    prompt_text="The bunny is eating a carrot",
) 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) when making this request.

```python
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`.

Configuring the HTTP client

You can directly override the httpx client to customize it for your use case, including:

- Support for proxies
- Custom transports
- Additional advanced functionality

```python
import httpx
from runwayml import RunwayML, DefaultHttpxClient

client = RunwayML(
    # Or use the `RUNWAYML_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. You can manually close the client using the `.close()` method if desired, or with a context manager that closes when exiting.

```python
from runwayml import RunwayML

with RunwayML() as client:
  # make requests here
  ...

# HTTP client is now closed
```

Versioning

This package generally follows SemVer 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 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:

```python
import runwayml
print(runwayml.__version__)
```

Requirements

Python 3.8 or higher.

Contributing

See the contributing documentation.