Runway API SDKs

Available SDKs

We provide SDKs as convenient helpers for interacting with our API. These SDKs use best practices and offer type safety, which helps to avoid errors and make it easier to write code.

Node.js

https://www.npmjs.com/package/@runwayml/sdk

The Node.js SDK includes TypeScript bindings. It is compatible with Node 18 and up, and can be installed with `npm`, `yarn`, or `pnpm`.

Python

https://pypi.org/project/runwayml/

The Python SDK includes type annotations compatible with MyPy. It is compatible with Python 3.8 and up.

Generating content

You can create content using our API using the methods documented in the API reference. For instance, `POST /v1/text_to_image` accepts input and produces an image as output.

Each API endpoint for starting a generation is available as a member on the SDKs. Here is a mapping of the API endpoints to the SDK methods:

Node | Python

Operation	API endpoint	Node.js SDK method
Generate an image	`POST /v1/text_to_image`	`client.textToImage.create`
Generate a video	`POST /v1/image_to_video`	`client.imageToVideo.create`
Video upscale	`POST /v1/video_upscale`	`client.videoUpscale.create`
Character performance	`POST /v1/character_performance`	`client.characterPerformance.create`

Calling these methods will create a task. A task is a record of the generation operation. The response from the method will look like this:

```
{
  "id": "17f20503-6c24-4c16-944b-35dbbce2af2f"
}
```

The `id` field is the unique identifier for the task. You can use this ID to retrieve the task status and output from the `GET /v1/tasks/{id}` endpoint, which is available as the `tasks.retrieve` method on the SDKs.

Node | Python

```
import RunwayML from '@runwayml/sdk';
const client = new RunwayML();

const task = await client.tasks.retrieve('17f20503-6c24-4c16-944b-35dbbce2af2f');
console.log(task);
```

The response from the method will look like this:

```
{
  "id": "17f20503-6c24-4c16-944b-35dbbce2af2f",
  "status": "PENDING",
  "createdAt": "2024-06-27T19:49:32.334Z"
}
```

The API reference documents the statuses that a task can be in, along with the fields that are available for each status.

Tasks are processed asychronously. The `tasks.retrieve` method returns the current status of the task, which you can poll until the task has completed. The task will eventually transition to a `SUCCEEDED`, `CANCELED`, or `FAILED` status.

When polling, we recommend using an interval of 5 seconds or more. You should also add jitter, and handle non-200 responses with exponential backoff. Avoid using fixed interval polling (such as with JavaScript's `setInterval`) since latency from the API can cause the polling to be too frequent.

Built-in task polling

As a convenience, all SDK methods that return a task include a helper method that polls for the task output. This reduces the amount of code you need to write to wait for a task to complete.

Node | Python

The `waitForTaskOutput` method is present on the unawaited response from the `create` methods on `textToImage`, `imageToVideo`, and `videoUpscale`.

```
// ✅ Call `waitForTaskOutput` on the unawaited response from `create`
const imageTask = await client.textToImage
  .create({
    model: 'gen4_image',
    promptText: 'A beautiful sunset over a calm ocean',
    ratio: '1360:768',
  })
  .waitForTaskOutput();

console.log(imageTask.output[0]); // Print the URL of the generated image
```

```
// ✅ Getting the task ID for bookkeeping purposes

// Notice: no `await` here.
const imageTask = client.textToImage.create({
  model: 'gen4_image',
  promptText: 'A beautiful sunset over a calm ocean',
  ratio: '1360:768',
});

// Await the output of `create` to get the task ID
const taskId = (await imageTask).id;
console.log(taskId); // The task ID can be stored in your database, for instance.

// Wait for the task to complete. It is safe to await `waitForTaskOutput`
// after the output of `create` was awaited above.
const completedTask = await imageTask.waitForTaskOutput();

console.log(completedTask.output[0]); // Print the URL of the generated image
```

```
// ❌ If you await the response from `create`, the result not have access to
//  `waitForTaskOutput`.
const awaitedImageTask = await client.textToImage.create({
  model: 'gen4_image',
  promptText: 'A beautiful sunset over a calm ocean',
  ratio: '1360:768',
});
Property 'waitForTaskOutput' does not exist on type 'TextToImageCreateResponse'.
const taskOutput = await awaitedImageTask.waitForTaskOutput();
```

If the task fails (that is, its status becomes `FAILED`), a `TaskFailedError` will be thrown. You should handle this error appropriately.

```
import { TaskFailedError } from '@runwayml/sdk';
try {
  const imageTask = await client.textToImage
    .create({
      model: 'gen4_image',
      promptText: 'A beautiful sunset over a calm ocean',
      ratio: '1360:768',
    })
    .waitForTaskOutput();
} catch (error) {
  if (error instanceof TaskFailedError) {
    // `taskDetails` contains the output of the tasks.retrieve call.
    console.error('Task failed:', error.taskDetails);
  } else {
    throw error;
  }
}
```

The `waitForTaskOutput` method accepts an optional `options` parameter. This parameter can be used to specify a timeout and an `AbortSignal`.

```
const imageTask = await client.textToImage
  .create({
    model: 'gen4_image',
    promptText: 'A beautiful sunset over a calm ocean',
    ratio: '1360:768',
  })
  .waitForTaskOutput({
    // Wait up to 5 minutes for the task to complete
    timeout: 5 * 60 * 1000,
    // Abort the task if the request is cancelled
    abortSignal: myAbortSignal,
  });
```

By default, `waitForTaskOutput` will wait for ten minutes before timing out. Upon timeout, a `TaskTimedOutError` will be thrown. Pass `null` to `timeout` to wait indefinitely. Disabling the timeout is not recommended as it may cause your server to experience issues if your Runway API organization reaches its concurrency limit or if Runway experiences an outage.

It is recommended to use an `AbortSignal` to cancel polling if you are using `waitForTaskOutput` in the handler for an incoming request, such as on a web server. Here is an example of how to correctly integrate this into your application:

Express.js | Koa | Socket.io

```
const runway = new RunwayML();

app.post('/generate-image', (req, res) => {
  // Create an AbortController that triggers when the request is closed
  // unexpectedly.
  const abortController = new AbortController();
  req.on('close', () => {
    abortController.abort();
  });

  // 🚨 When performing a generation, be sure to add appropriate rate limiting
  // and other safeguards to prevent abuse.
  try {
    const imageTask = await runway.textToImage
      .create({
        model: 'gen4_image',
        promptText: req.body.prompt,
        ratio: '1360:768',
      })
      .waitForTaskOutput({ abortSignal: abortController.signal });

    res.send(imageTask.output[0]);
  } catch (error) {
    if (error instanceof TaskFailedError) {
      res.status(500).send('Task failed');
    } else {
      throw error;
    }
  }
});
```

> **Danger**
>
> Triggering the `abortSignal` passed to `waitForTaskOutput` or hitting the passed `timeout` will not cancel the task. Cancelling the task must be done by invoking the cancellation endpoint.

In addition to the methods that create new tasks, the `tasks.retrieve` method also returns a promise with a `waitForTaskOutput` method. This method is equivalent to the `waitForTaskOutput` method on the unawaited response from the `create` methods.

```
const task = await client.tasks
  .retrieve('17f20503-6c24-4c16-944b-35dbbce2af2f')
  .waitForTaskOutput();
console.log(task.output[0]);
```

This is useful if you'd like to create a task in one request and wait for its output in another request, or for handling the case where the client disconnected before the task completed.

Be aware that you must still add error handling for `TaskFailedError` and `TaskTimeoutError` when using this method.