Predicted Outputs

Copy page

Reduce latency for model responses where much of the response is known ahead of time.

Predicted Outputs enable you to speed up API responses from Chat Completions when many of the output tokens are known ahead of time. This is most common when you are regenerating a text or code file with minor modifications. You can provide your prediction using the `prediction` request parameter in Chat Completions.

Predicted Outputs are available today using the latest `gpt-4o` and `gpt-4o-mini` models. Read on to learn how to use Predicted Outputs to reduce latency in your applicatons.

Code refactoring example

Predicted Outputs are particularly useful for regenerating text documents and code files with small modifications. Let's say you want the GPT-4o model to refactor a piece of TypeScript code, and convert the `username` property of the `User` class to be `email` instead:

```
1 class User {
2   firstName: string = "";
3   lastName: string = "";
4   username: string = "";
5 }
6
7 export default User;
```

Most of the file will be unchanged, except for line 4 above. If you use the current text of the code file as your prediction, you can regenerate the entire file with lower latency. These time savings add up quickly for larger files.

Below is an example of using the `prediction` parameter in our SDKs to predict that the final output of the model will be very similar to our original code file, which we use as the prediction text.

Refactor a TypeScript class with a Predicted Output — javascript

```
1  import OpenAI from "openai";
2
3  const code = `
4  class User {
5    firstName: string = "";
6    lastName: string = "";
```

```
  username: string = "";
}

export default User;
`.trim();

const openai = new OpenAI();

const refactorPrompt = `
Replace the "username" property with an "email" property. Respond only
with code, and with no markdown formatting.
`;

const completion = await openai.chat.completions.create({
  model: "gpt-4o",
  messages: [
    {
      role: "user",
      content: refactorPrompt
    },
    {
      role: "user",
      content: code
    }
  ],
  store: true,
  prediction: {
    type: "content",
    content: code
  }
});

// Inspect returned data
console.log(completion);
console.log(completion.choices[0].message.content);
```

In addition to the refactored code, the model response will contain data that looks something like this:

```
{
  id: 'chatcmpl-xxx',
  object: 'chat.completion',
  created: 1730918466,
  model: 'gpt-4o-2024-08-06',
  choices: [ /* ...actual text response here... */],
  usage: {
    prompt_tokens: 81,
    completion_tokens: 39,
    total_tokens: 120,
    prompt_tokens_details: { cached_tokens: 0, audio_tokens: 0 },
    completion_tokens_details: {
      reasoning_tokens: 0,
      audio_tokens: 0,
      accepted_prediction_tokens: 18,
```

```
      rejected_prediction_tokens: 10
    }
  },
  system_fingerprint: 'fp_159d8341cc'
}
```

Note both the `accepted_prediction_tokens` and `rejected_prediction_tokens` in the `usage` object. In this example, 18 tokens from the prediction were used to speed up the response, while 10 were rejected.

Note that any rejected tokens are still billed like other completion tokens generated by the API, so Predicted Outputs can introduce higher costs for your requests.

Streaming example

The latency gains of Predicted Outputs are even greater when you use streaming for API responses. Here is an example of the same code refactoring use case, but using streaming in the OpenAI SDKs instead.

Predicted Outputs with streaming — javascript

```javascript
import OpenAI from "openai";

const code = `
class User {
  firstName: string = "";
  lastName: string = "";
  username: string = "";
}

export default User;
`.trim();

const openai = new OpenAI();

const refactorPrompt = `
Replace the "username" property with an "email" property. Respond only
with code, and with no markdown formatting.
`;

const completion = await openai.chat.completions.create({
  model: "gpt-4o",
  messages: [
    {
      role: "user",
      content: refactorPrompt
    },
    {
      role: "user",
      content: code
    }
```

```
    ],
    store: true,
    prediction: {
      type: "content",
      content: code
    },
    stream: true
  });

  // Inspect returned data
  for await (const chunk of stream) {
    process.stdout.write(chunk.choices[0]?.delta?.content || "");
  }
```

Position of predicted text in response

When providing prediction text, your prediction can appear anywhere within the generated response, and still provide latency reduction for the response. Let's say your predicted text is the simple Hono server shown below:

```
import { serveStatic } from "@hono/node-server/serve-static";
import { serve } from "@hono/node-server";
import { Hono } from "hono";

const app = new Hono();

app.get("/api", (c) => {
  return c.text("Hello Hono!");
});

// You will need to build the client code first `pnpm run ui:build`
app.use(
  "/*",
  serveStatic({
    rewriteRequestPath: (path) => `./dist${path}`,
  })
);

const port = 3000;
console.log(`Server is running on port ${port}`);

serve({
  fetch: app.fetch,
  port,
});
```

You could prompt the model to regenerate the file with a prompt like:

```
Add a get route to this application that responds with
the text "hello world". Generate the entire application
```

```
file again with this route added, and with no other
markdown formatting.
```

The response to the prompt might look something like this:

```
import { serveStatic } from "@hono/node-server/serve-static";
import { serve } from "@hono/node-server";
import { Hono } from "hono";

const app = new Hono();

app.get("/api", (c) => {
  return c.text("Hello Hono!");
});

app.get("/hello", (c) => {
  return c.text("hello world");
});

// You will need to build the client code first `pnpm run ui:build`
app.use(
  "/*",
  serveStatic({
    rewriteRequestPath: (path) => `./dist${path}`,
  })
);

const port = 3000;
console.log(`Server is running on port ${port}`);

serve({
  fetch: app.fetch,
  port,
});
```

You would still see accepted prediction tokens in the response, even though the prediction text appeared both before and after the new content added to the response:

```
{
  id: 'chatcmpl-xxx',
  object: 'chat.completion',
  created: 1731014771,
  model: 'gpt-4o-2024-08-06',
  choices: [ /* completion here... */],
  usage: {
    prompt_tokens: 203,
    completion_tokens: 159,
    total_tokens: 362,
    prompt_tokens_details: { cached_tokens: 0, audio_tokens: 0 },
    completion_tokens_details: {
      reasoning_tokens: 0,
      audio_tokens: 0,
      accepted_prediction_tokens: 60,
      rejected_prediction_tokens: 0
```

```
      }
    },
    system_fingerprint: 'fp_9ee9e968ea'
  }
```

This time, there were no rejected prediction tokens, because the entire content of the file we predicted was used in the final response. Nice! 🔥

Limitations

When using Predicted Outputs, you should consider the following factors and limitations.

- Predicted Outputs are only supported with the GPT-4o and GPT-4o-mini series of models.
- When providing a prediction, any tokens provided that are not part of the final completion are still charged at completion token rates. See the `rejected_prediction_tokens` property of the `usage` object to see how many tokens are not used in the final response.
- The following API parameters are not supported when using Predicted Outputs:
  - `n`: values higher than 1 are not supported
  - `logprobs`: not supported
  - `presence_penalty`: values greater than 0 are not supported
  - `frequency_penalty`: values greater than 0 are not supported
  - `audio`: Predicted Outputs are not compatible with audio inputs and outputs
  - `modalities`: Only `text` modalities are supported
  - `max_completion_tokens`: not supported
  - `tools`: Function calling is not currently supported with Predicted Outputs