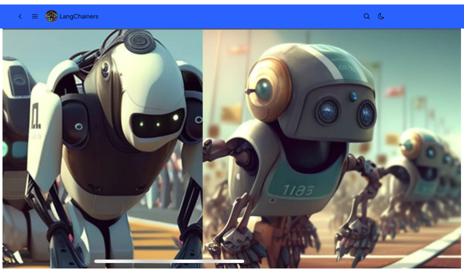


Getting Started with LangChain.js

A Step-by-Step Tutorial

LangChainers · Feb 26, 2023 · 12 min read

(This tutorial is intentionally oversimplified and overly instructive with the objective of helping make AI more approachable. Examples are from the LangChain library with more "hand holding" for those who need help getting started)

Pre-requisites

Node.js

This tutorial requires that you have Node.js installed. Node.js is the Javascript compiler/interpreter that is used to run Javascript applications. If you do not have Node.js installed, install Node.js from the Node.js installation site here

Install the NPM package and project manager

If you do not have NPM already installed, you can install it from here

Setup

Create the tutorial project folder

```
$ mkdir langchainjs-tutorial-1
$ cd langchainjs-tutorial-1
```

Initialize your project as an `npm` project and create the package.json project configuration file
(The es6 flag tells npm to create a package.json with the "type": "module" setting.
The -y flag tells npm *to accept the defaults)*

```
$ npm init es6 -y
```

Install the TypeScript Node package. Configures Node to run TypeScript code.

```
$ npm install typescript
```

Configure capability to run Typescript code using Node commands

```
$ npm install -D ts-node
```

Add the type definitions for Node to run TypeScript

```
$ npm install @types/node
```

Create the tsconfig.json that contains the information required to compile TypeScript to JavaScript

```
$ npx tsc --init --rootDir src --outDir ./dist --esModuleInterop --lib ES2020 --target ES2020 --module nodenext --noImplicitAny true
```

Add build and run scripts to package.json. Replace "scripts" entry with the JSON below. If you do not have a "scripts" entry in your package.json, place the JSON below above *dependencies* (or *devDependencies)*

```
"scripts": {
    "build": "tsc",
    "start": "node ./dist/app.js",
    "dev": "ts-node --esm ./src/app.ts"
},
```

Create an app.ts source file in the src folder. Add code to it to display a test string.
(if the "echo" command does not work for you, simply create app.ts in the src folder and cut and paste the "console.log..." statement)

```
$ mkdir src
$ echo "console.log('Welcome to the LangChain.js tutorial by LangChainers.')" > src/app.ts
```

Perform a test to verify that the tutorial project can build and run using "yarn"

```
$ npm run build
$ npm run start
```

You can also use `$ npm run dev` for a single command that executes both of the above.

If everything works, you are ready for some LanngChain.js. LET'S GO!!

Installation

Install the LangChain.js package

```
$ npm install langchain
```

Install the OpenAI package

```
$ npm install openai
```

Install the dotenv package for managing environment variables including your OpenAI API key

```
$ npm install dotenv
```

Create a file named `.env` in the project directory and add your OpenAI API Key to the file as shown below
(if you do not have an OpenAI API Key, go to your OpenAI API Key page, then cut and paste the OpenAI API Key below)

```
OPENAI_API_KEY=<remove the < and > and put your OpenAI API key here>
```

Now let's walk through some LangChain.js modules

LangChain Modules

LLMs

LLMs (Large Language Models) are the core of LangChain functionality. The LLM module is simply a wrapper around different LLMs. This wrapper makes it simple for LangChain.js developers to communicate with different LLMs using a single interface, without having to worry about the difference between the different LLMs.

Copy the following code into src/app.ts

```
//Import the OpenAPI Large Language Model (you can import other models here eg. Cohere)
import { OpenAI } from "langchain/llms";

//Load environment variables (populate process.env from .env file)
import * as dotenv from "dotenv";
dotenv.config();

export const run = async () => {

    //Instantiante the OpenAI model
    //Pass the "temperature" parameter which controls the RANDOMNESS of the model's output. A lower temperature will result in more predictable output, while a higher temperature will result in more random output. The temperature parameter is set between 0 and 1, with 0 being the most predictable and 1 being the most random
    const model = new OpenAI({ temperature: 0.9 });

    //Calls out to the model's (OpenAI's) endpoint passing the prompt. This call returns a string
    const res = await model.call(
        "What would be a good company name a company that makes colorful socks?"
    );
    console.log({ res });
};

run();
```

Execute the code with the following command

```
$ npm run dev
```

Result of the execution
(NOTE: The result you will get may differ from what you see below. It is, after all, AI)

```
{ res: '\n\nSocktastic!' }
```

We will cover LLMs in more detail in future tutorials. You can find API documentation for LLMs here

Prompt Template

A prompt template is an object that is responsible for constructing the final prompt to pass to an LLM. The same object can be reused with different data. The data can consist of the prompt ie. the input to the language model and user input data.

Copy the following code into src/app.ts

```
//Import the PromptTemplate module
import { PromptTemplate } from "langchain/prompts";

export const run = async () => {
    //Create the template. The template is actually a "parameterized prompt". A "parameterized prompt" is a prompt in which the input parameter names are used and the parameter values are supplied from external input
    const template = "What is a good name for a company that makes {product}?";

    //Instantiate "PromptTemplate" passing the prompt template string initialized above and a list of variable names the final prompt template will expect
    const prompt = new PromptTemplate({ template, inputVariables: ["product"] });

    //Create a new prompt from the combination of the template and input variables. Pass the value for the variable name that was sent in the "inputVariables" list passed to "PromptTemplate" initialization call
    const res = prompt.format({ product: "colorful socks" });
  console.log({ res });
};

run();
```

Execute the code with the following command

```
$ npm run dev
```

Result of the execution

The result is showing you that {product} in the "template" string has been replaced by the value "colorful socks" that was passed on the "prompt.format(..)" call.

```
{ res: 'What is a good name for a company that makes colorful socks?' }
```

We will cover Prompts and Prompt Templates in more detail in future tutorials. You can find API documentation for Prompt Templates here

Chains

Chains enable LangChain.js developers to combine multiple components together to create a functional application.

The following code will show an example of a chain that takes user input, formats it using PromptTemplate and then passes the formatted response to an LLM.

Copy the following code into src/app.ts

```
//Import the OpenAPI Large Language Model (you can import other models here eg. Cohere)
import { OpenAI } from "langchain/llms";

//Import the PromptTemplate module
import { PromptTemplate } from "langchain/prompts";

//Import the Chains module
import { LLMChain } from "langchain/chains";

//Load environment variables (populate process.env from .env file)
import * as dotenv from "dotenv";
dotenv.config();

export const run = async () => {
    //Instantiante the OpenAI model
    //Pass the "temperature" parameter which controls the RANDOMNESS of the model's output. A lower temperature will result in more predictable output, while a higher temperature will result in more random output. The temperature parameter is set between 0 and 1, with 0 being the most predictable and 1 being the most random
    const model = new OpenAI({ temperature: 0.9 });

    //Create the template. The template is actually a "parameterized prompt". A "parameterized prompt" is a prompt in which the input parameter names are used and the parameter values are supplied from external input
    const template = "What is a good name for a company that makes {product}?";

    //Instantiate "PromptTemplate" passing the prompt template string initialized above and a list of variable names the final prompt template will expect
    const prompt = new PromptTemplate({ template, inputVariables: ["product"] });

    //Instantiate LLMChain, which consists of a PromptTemplate and an LLM. Pass the result from the PromptTemplate and the OpenAI LLM model
    const chain = new LLMChain({ llm: model, prompt });

    //Run the chain. Pass the value for the variable name that was sent in the "inputVariables" list passed to "PromptTemplate" initialization call
    const res = await chain.call({ product: "colorful socks" });
    console.log({ res });
};

run();
```

Execute the code with the following command

```
$ npm run dev
```

Result of the execution

```
$ { res: { text: '\n\nSplash-a-Sox!' } }
```

We will cover Chains in more detail in future tutorials. You can find API documentation for Chains here

Agents

"Some applications will require not just a predetermined chain of calls to LLMs/other tools, but potentially an unknown chain that depends on the user input. In these types of chains, there is an "agent" which has access to a suite of tools. Depending on the user input, the agent can then decide which, if any, of these tools to call." - LangChain

Unlike "Chains" that are executed in a predetermined order, "Agents" use an LLM to determine the actions to take and the order in which the actions are executed.

In the following example, we will create an agent that uses a search tool and a calculator tool.

Install Search Tool

Install the SerpApi packange. SerpApi is a real-time API to access Google search results

```
$ npm install serpapi
```

Add your SerpApi API Key to the `.env` file as shown below

(if you do not have a SerpAPI API Key, go to your SerpApi API Key page, then cut and paste the SerpApi API Key below)

```
SERPAPI_API_KEY=<remove the < and > and put your SerpApi API key here>
```

Copy the following code into src/app.ts

```
//Import the OpenAPI Large Language Model (you can import other models here eg. Cohere)
import { OpenAI } from "langchain/llms";

//Import the agent executor module
import { initializeAgentExecutor } from "langchain/agents";

//Import the SerpAPI and Calculator tools
import { SerpAPI, Calculator } from "langchain/tools";

//Load environment variables (populate process.env from .env file)
import * as dotenv from "dotenv";
dotenv.config();

export const run = async () => {

    //Instantiante the OpenAI model
    //Pass the "temperature" parameter which controls the RANDOMNESS of the model's output. A lower temperature will result in more predictable output, while a higher temperature will result in more random output. The temperature parameter is set between 0 and 1, with 0 being the most predictable and 1 being the most random
    const model = new OpenAI({ temperature: 0 });

    //Create a list of the instatiated tools
    const tools = [new SerpAPI(), new Calculator()];

    //Construct an agent from an LLM and a list of tools
    //"zero-shot-react-description" tells the agent to use the ReAct framework to determine which tool to use. The ReAct framework determines which tool to use based solely on the tool’s description. Any number of tools can be provided. This agent requires that a description is provided for each tool.
    const executor = await initializeAgentExecutor(
    tools,
    model,
    "zero-shot-react-description"
    );
    console.log("Loaded agent.");

    //Specify the prompt
    const input =
    "Who is Beyonce's husband?" +
    " What is his current age raised to the 0.23 power?";
    console.log(`Executing with input "${input}"...`);

    //Run the agent
    const result = await executor.call({ input });

    console.log(`Got output ${result.output}`);
};

run();
```

Execute the code with the following command

```
$ npm run dev
```

Result of the execution

```
Loaded agent.
Executing with input "Who is Beyonce's husband? What is his current age raised to the 0.23 power?"...
Got output Jay-Z is 53 years old and his age raised to the 0.23 power is 2.4922032039503494.
```

We will cover Agents in more detail in future tutorials. You can find API documentation for Agents here

Memory

Chains and agents are stateless by default. Memory persists state between calls of chain or agent, thereby enabling true conversations. Practical conversations are those that can recall previous conversations or elements of previous conversations (eg. facts given in an earlier part of the conversation). This is useful for ChatBots.

The following code will show an example of memory.

Copy the following code into src/app.ts

```
//Import the OpenAPI Large Language Model (you can import other models here eg. Cohere)
import { OpenAI } from "langchain/llms";

//Import the BufferMemory module
import { BufferMemory } from "langchain/memory";

//Import the Chains module
import { LLMChain } from "langchain/chains";

//Import the PromptTemplate module
import { PromptTemplate } from "langchain/prompts";

//Load environment variables (populate process.env from .env file)
import * as dotenv from "dotenv";
dotenv.config();

export const run = async () => {

    //Instantiate the BufferMemory passing the memory key for storing state
    const memory = new BufferMemory({ memoryKey: "chat_history" });

    //Instantiante the OpenAI model
    //Pass the "temperature" parameter which controls the RANDOMNESS of the model's output. A lower temperature will result in more predictable output, while a higher temperature will result in more random output. The temperature parameter is set between 0 and 1, with 0 being the most predictable and 1 being the most random
    const model = new OpenAI({ temperature: 0.9 });

    //Create the template. The template is actually a "parameterized prompt". A "parameterized prompt" is a prompt in which the input parameter names are used and the parameter values are supplied from external input
    //Note the input variables {chat_history} and {input}
    const template = `The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.
    Current conversation:
    {chat_history}
    Human: {input}
    AI:`;

    //Instantiate "PromptTemplate" passing the prompt template string initialized above
    const prompt = PromptTemplate.fromTemplate(template);

    //Instantiate LLMChain, which consists of a PromptTemplate, an LLM and memory.
    const chain = new LLMChain({ llm: model, prompt, memory });

    //Run the chain passing a value for the {input} variable. The result will be stored in {chat_history}
    const res1 = await chain.call({ input: "Hi! I'm Morpheus." });
    console.log({ res1 });

    //Run the chain again passing a value for the {input} variable. This time, the response from the last run ie. the  value in {chat_history} will alo be passed as part of the prompt
    const res2 = await chain.call({ input: "What's my name?" });
    console.log({ res2 });

    //BONUS!!
    const res3 = await chain.call({ input: "Which epic movie was I in and who was my protege?" });
    console.log({ res3 });
};
```

```
run();
```

Execute the code with the following command

```
$ npm run dev
```

Result of the execution

(The response you will get may be slightly different in wording from what you see below)

```
{
  res1: {
    text: " Hi Morpheus! It's nice to meet you. I'm an AI created to answer your questions. What can I do for you today?"
  }
}
{
  res2: {
    text: ' Your name is Morpheus. Is there anything else I can help you with?'
  }
}
{
  res3: {
    text: 'You were in The Matrix, and your protege was Neo. Would you like to talk about something else?'
  }
}
```

We will cover Memory in more detail in future tutorials. You can find API documentation for Memory here

We hope this Step-By-Step Tutorial was helpful in getting you started with LangChain.js. We will be digging deeper into the individual modules and use cases in upcoming tutorials. We cant wait. We love LangChain. STAY TUNED!!

Pro Teams Pricing Documentation

npm

Sign Up Sign In

Search packages Search

openai TS

3.2.1 • Public • Published a month ago

Readme

Code Beta

2 Dependencies

437 Dependents

25 Versions

OpenAI Node.js Library

The OpenAI Node.js library provides convenient access to the OpenAI API from Node.js applications. Most of the code in this library is generated from our OpenAPI specification.

Important note: this library is meant for server-side usage only, as using it in client-side browser code will expose your secret API key. See here for more details.

Installation

```
$ npm install openai
```

Usage

The library needs to be configured with your account's secret key, which is available on the website. We recommend setting it as an environment variable. Here's an example of initializing the library with the API key loaded from an environment variable and creating a completion:

```
const { Configuration, OpenAIApi } = require("openai");

const configuration = new Configuration({
  apiKey: process.env.OPENAI_API_KEY,
});
const openai = new OpenAIApi(configuration);

const completion = await openai.createCompletion({
  model: "text-davinci-003",
  prompt: "Hello world",
});
console.log(completion.data.choices[0].text);
```

Check out the **full API documentation** for examples of all the available functions.

Request options

All of the available API request functions additionally contain an optional final parameter where you can pass custom **axios request options**, for example:

```
const completion = await openai.createCompletion(
  {
    model: "text-davinci-003",
    prompt: "Hello world",
  },
  {
    timeout: 1000,
    headers: {
      "Example-Header": "example",
    },
  }
);
```

Error handling

API requests can potentially return errors due to invalid inputs or other issues. These errors can be handled with a `try...catch` statement, and the error details can be found in either `error.response` or `error.message`:

```
try {
  const completion = await openai.createCompletion({
    model: "text-davinci-003",
    prompt: "Hello world",
  });
  console.log(completion.data.choices[0].text);
} catch (error) {
  if (error.response) {
    console.log(error.response.status);
    console.log(error.response.data);
  } else {
    console.log(error.message);
  }
}
```

Streaming completions

Streaming completions (`stream=true`) are not natively supported in this package yet, but **a workaround exists** if needed.

Upgrade guide

All breaking changes for major version releases are listed below.

3.0.0

- The function signature of `createCompletion(engineId, params)` changed to `createCompletion(params)`. The value previously passed in as the `engineId` argument should now be passed in as `model` in the params object (e.g. `createCompletion({ model: "text-davinci-003", ... })`)
- Replace any `createCompletionFromModel(params)` calls with `createCompletion(params)`

Thanks

Thank you to ceifa for creating and maintaining the original unofficial `openai` npm package before we released this official library! ceifa's original package has been renamed to gpt-x.

Keywords

openai open ai gpt-3 gpt3

Install

```
> npm i openai
```

Repository

github.com/openai/openai-node

Homepage

github.com/openai/openai-node#readme

Weekly Downloads

449,234

Version	License
3.2.1	MIT

Unpacked Size	Total Files
480 kB	26

Last publish

a month ago

Collaborators

>_Try on RunKit

⚑Report malware

Support

Help

Advisories

Status

Contact npm

Company

About

Blog

Press

Terms & Policies

Policies

Terms of Use

Code of Conduct

Privacy

Node.JS Client

This page provides installation instructions, usage examples, and a reference for the Pinecone Node.JS client.

⚠ **Warning**

This is a **public preview** ("Beta") client. Test thoroughly before using this client for production workloads. No SLAs or technical support commitments are provided for this client. Expect potential breaking changes in future releases.

Getting Started

Installation

Use the following shell command to install the Node.JS client for use with Node.JS versions 17 and above:

Shell

```
npm install @pinecone-database/pinecone
```

Alternatively, you can install Pinecone with Yarn:

Shell

```
yarn add @pinecone-database/pinecone
```

Usage

Initialize the client

To initialize the client, instantiate the `PineconeClient` class and call the `init` method. The `init` method takes an object with the `apiKey` and `environment` properties:

JavaScript

```
import { PineconeClient } from "@pinecone-database/pinecone";

const pinecone = new PineconeClient();
await pinecone.init({
  environment: "YOUR_ENVIRONMENT",
  apiKey: "YOUR_API_KEY",
});
```

Create index

The following example creates an index without a metadata configuration. By default, Pinecone indexes all metadata.

JavaScript

```
await pinecone.createIndex({
  createRequest: {
    name: "example-index",
    dimension: 1024,
  },
});
```

The following example creates an index that only indexes the "color" metadata field. Queries against this index cannot filter based on any other metadata field.

JavaScript

```
await pinecone.createIndex({
  createRequest: {
    name: "example-index-2",
    dimension: 1024,
    metadata_config: {
      indexed: ["color"],
    },
  },
});
```

List indexes

The following example logs all indexes in your project.

JavaScript

```
const indexesList = await pinecone.listIndexes();
```

Describe index

The following example logs information about the index `example-index`.

JavaScript

```
const indexDescription = await pinecone.describeIndex({
  indexName: "example-index",
});
```

Delete index

The following example deletes `example-index`.

JavaScript

```
await pinecone.deleteIndex({
  indexName: "example-index",
});
```

Scale replicas

The following example sets the number of replicas and pod type for `example-index`.

JavaScript

```
await pinecone.configureIndex({
  indexName: "example-index",
  patchRequest: {
    replicas: 2,
    podType: "p2",
  },
});
```

Describe index statistics

The following example returns statistics about the index `example-index`.

JavaScript

```
const index = pinecone.Index("example-index");
const indexStats = index.describeIndexStats({
  describeIndexStatsRequest: {
    filter: {},
  },
});
```

Upsert vectors

The following example upserts vectors to `example-index`.

JavaScript

```javascript
const index = pinecone.Index("example-index");
const upsertRequest = {
  vectors: [
    {
      id: "vec1",
      values: [0.1, 0.2, 0.3, 0.4],
      metadata: {
        genre: "drama",
      },
    },
    {
      id: "vec2",
      values: [0.2, 0.3, 0.4, 0.5],
      metadata: {
        genre: "action",
      },
    },
  ],
  namespace: "example-namespace",
};
const upsertResponse = await index.upsert({ upsertRequest });
```

Query an index

The following example queries the index `example-index` with metadata filtering.

JavaScript

```javascript
const index = pinecone.Index("example-index");
const queryRequest = {
  vector: [0.1, 0.2, 0.3, 0.4],
  topK: 10,
  includeValues: true,
  includeMetadata: true,
  filter: {
    genre: { $in: ["comedy", "documentary", "drama"] },
  },
  namespace: "example-namespace",
};
const queryResponse = await index.query({ queryRequest });
```

Delete vectors

The following example deletes vectors by ID.

JavaScript

```
const index = pinecone.Index("example-index");
await index.delete1({
  ids: ["vec1", "vec2"],
  namespace: "example-namespace",
});
```

Fetch vectors

The following example fetches vectors by ID.

JavaScript

```
const index = pinecone.Index("example-index");
const fetchResponse = await index.fetch({
  ids: ["vec1", "vec2"],
  namespace: "example-namespace",
});
```

Update vectors

The following example updates vectors by ID.

JavaScript

```
const index = pinecone.Index("example-index");
const updateRequest = {
  id: "vec1",
  values: [0.1, 0.2, 0.3, 0.4],
  setMetadata: { genre: "drama" },
  namespace: "example-namespace",
};
const updateResponse = await index.update({ updateRequest });
```

Create collection

The following example creates the collection `example-collection` from `example-index`.

JavaScript

```
const createCollectionRequest = {
  name: "example-collection",
  source: "example-index",
};

await pinecone.createCollection({
  createCollectionRequest,
});
```

List collections

The following example returns a list of the collections in the current project.

JavaScript

```
const collectionsList = await pinecone.listCollections();
```

Describe a collection

The following example returns a description of the collection `example-collection`.

JavaScript

```
const collectionDescription = await pinecone.describeCollection({
  collectionName: "example-collection",
});
```

Delete a collection

The following example deletes the collection `example-collection`.

JavaScript

```
await pinecone.deleteCollection({
  collectionName: "example-collection",
});
```

Reference

For the REST API or other clients, see the API reference.

init()

`pinecone.init(configuration: PineconeClientConfiguration)`

Initialize the Pinecone client.

Parameters	Type	Description
`configuration`	PineconeClientConfiguration	The configuration for the Pinecone client.

Types

PineconeClientConfiguration

Parameters	Type	Description
`apiKey`	string	The API key for the Pinecone service.
`environment`	string	The cloud environment of your Pinecone project.

Example:

JavaScript

```javascript
import { PineconeClient } from "@pinecone-database/pinecone";
const pinecone = new PineconeClient();
await pinecone.init({
  apiKey: "YOUR_API_KEY",
  environment: "YOUR_ENVIRONMENT",
});
```

configureIndex()

`pinecone.configure_index(indexName: string, patchRequest?: PatchRequest)`

Configure an index to change pod type and number of replicas.

Parameters	Type	Description
`requestParameters`	ConfigureIndexRequest	Index configuration parameters.

Types

ConfigureIndexRequest

Parameters	Type	Description
`indexName`	string	The name of the index.
`patchRequest`	PatchRequest	(Optional) Patch request parameters.

PatchRequest

Parameters	Type	Description
`replicas`	number	(Optional) The number of replicas to configure for this index.
`podType`	string	(Optional) The new pod type for the index. One of `s1` , `p1` , or `p2` appended with `.` and one of `x1` , `x2` , `x4` , or `x8` .

Example:

JavaScript

```
const newNumberOfReplicas = 4;
const newPodType = "s1.x4";
await pinecone.configureIndex({
  indexName: "example-index",
  patchRequest: {
    replicas: newNumberOfReplicas,
    podType: newPodType,
  },
});
```

createCollection()

`pinecone.createCollection(requestParameters: CreateCollectionOperationRequest)`

Create a collection from an index.

Parameters	Type	Description
`requestParameters`	CreateCollectionOperationRequest	Create collection operation wrapper

Types

CreateCollectionOperationRequest

Parameters	Type	Description
`createCollectionRequest`	CreateCollectionRequest	Collection request parameters.

CreateCollectionRequest

Parameters	Type	Description
`name`	string	The name of the collection to be created.

Parameters	Type	Description
`source`	string	The name of the source index to be used as the source for the collection.

Example:

JavaScript

```
await pinecone.createCollection({
  createCollectionRequest: {
    name: "example-collection",
    source: "example-index",
  },
});
```

createIndex()

`pinecone.createIndex(requestParameters?: CreateIndexRequest)`

Create an index.

Parameters	Type	Description
`requestParameters`	CreateIndexRequest	Create index operation wrapper

Types

CreateIndexRequest

Parameters	Type	Description
`createRequest`	CreateRequest	Create index request parameters

CreateRequest

Parameters	Type	Description
`name`	str	The name of the index to be created. The maximum length is 45 characters.
`dimension`	integer	The dimensions of the vectors to be inserted in the index.
`metric`	str	(Optional) The distance metric to be used for similarity search: 'euclidean', 'cosine', or 'dotproduct'.

Parameters	Type	Description
`pods`	int	(Optional) The number of pods for the index to use, including replicas.
`replicas`	int	(Optional) The number of replicas.
`pod_type`	str	(Optional) The new pod type for the index. One of `s1` , `p1` , or `p2` appended with `.` and one of `x1` , `x2` , `x4` , or `x8` .
`metadata_config`	object	(Optional) Configuration for the behavior of Pinecone's internal metadata index. By default, all metadata is indexed; when metadata_config is present, only specified metadata fields are indexed. To specify metadata fields to index, provide a JSON object of the following form: `{"indexed": ["example_metadata_field"]}`
`source_collection`	str	(Optional) The name of the collection to create an index from.

Example:

JavaScript

```javascript
// The following example creates an index without a metadata
// configuration. By default, Pinecone indexes all metadata.
await pinecone.createIndex({
  createRequest: {
    name: "pinecone-index",
    dimension: 1024,
  },
});

// The following example creates an index that only indexes
// the 'color' metadata field. Queries against this index
// cannot filter based on any other metadata field.

await pinecone.createIndex({
  createRequest: {
    name: "example-index-2",
    dimension: 1024,
    metadata_config: {
      indexed: ["color"],
    },
  },
});
```

deleteCollection()

`pinecone.deleteCollection(requestParameters: DeleteCollectionRequest)`

Delete an existing collection.

Types

Parameters	Type	Description
`requestParameters`	DeleteCollectionRequest	Delete collection request parameters

DeleteCollectionRequest

Parameters	Type	Description
`collectionName`	string	The name of the collection to delete.

Example:

JavaScript

```
await pinecone.deleteCollection({
  collectionName: "example-collection",
});
```

deleteIndex()

`pinecone.deleteIndex(requestParameters: DeleteIndexRequest)`

Delete an index.

Types

Parameters	Type	Description
`requestParameters`	DeleteIndexRequest	Delete index request parameters

DeleteIndexRequest

Parameters	Type	Description
`indexName`	string	The name of the index to delete.

Example:

JavaScript

```
await pinecone.deleteIndex({
  indexName: "example-index",
});
```

describeCollection()

`pinecone.describeCollection(requestParameters: DescribeCollectionRequest)`

Get a description of a collection.

Types

Parameters	Type	Description
`requestParameters`	DescribeCollectionRequest	Describe collection request parameters

DescribeCollectionRequest

Parameters	Type	Description
`collectionName`	string	The name of the collection.

Example:

JavaScript

```
const collectionDescription = await pinecone.describeCollection({
  collectionName: "example-collection",
});
```

Return:

- `collectionMeta` : `object` Configuration information and deployment status of the collection.
  - `name` : `string` The name of the collection.
  - `size` : `integer` The size of the collection in bytes.
  - `status` : `string` The status of the collection.

describeIndex()

`pinecone.describeIndex(requestParameters: DescribeIndexRequest)`

Get a description of an index.

Types

Parameters	Type	Description
`requestParameters`	DescribeIndexRequest	Describe index request parameters

DescribeIndexRequest

Parameters	Type	Description
`indexName`	string	The name of the index.

Types

Returns:

- `database` : `object`
- `name` : `string` The name of the index.
- `dimension` : `integer` The dimensions of the vectors to be inserted in the index.
- `metric` : `string` The distance metric used for similarity search: 'euclidean', 'cosine', or 'dotproduct'.
- `pods` : `integer` The number of pods the index uses, including replicas.
- `replicas` : `integer` The number of replicas.
- `pod_type` : `string` The pod type for the index. One of `s1` , `p1` , or `p2` appended with `.` and one of `x1` , `x2` , `x4` , or `x8` .
- `metadata_config` : `object` Configuration for the behavior of Pinecone's internal metadata index. By default, all metadata is indexed; when metadata_config is present, only specified metadata fields are indexed. To specify metadata fields to index, provide a JSON object of the following form: `{"indexed": ["example_metadata_field"]}`
- `status` : `object`
- `ready` : `boolean` Whether the index is ready to serve queries.
- `state` : `string` One of `Initializing` , `ScalingUp` , `ScalingDown` , `Terminating` , or `Ready` .

Example:

JavaScript

```javascript
const indexDescription = await pinecone.describeIndex({
  indexName: "example-index",
});
```

listCollections

`pinecone.listCollections()`

Return a list of the collections in your project.

Example:

JavaScript

```
const collections = await pinecone.listCollections();
```

Returns:

- `array` of `strings` The names of the collections in your project.

listIndexes

`pinecone.listIndexes()`

Return a list of your Pinecone indexes.

Returns:

- `array` of `strings` The names of the indexes in your project.

Example:

JavaScript

```
const indexesList = await pinecone.listIndexes();
```

Index()

`pinecone.Index(indexName: string)`

Construct an Index object.

Parameters	Type	Description
`indexName`	string	The name of the index.

Example:

JavaScript

```
const index = pinecone.Index("example-index");
```

Index.delete1()

`index.delete(requestParameters: Delete1Request)`

Delete items by their ID from a single namespace.

Parameters	Type	Description
`requestParameters`	Delete1Request	Delete request parameters

Types

Delete1Request

Parameters	Type	Description
`ids`	Array	(Optional) The IDs of the items to delete.
`deleteAll`	boolean	(Optional) Indicates that all vectors in the index namespace should be deleted.
`namespace`	str	(Optional) The namespace to delete vectors from, if applicable.

Types

Example:

JavaScript

```javascript
await index.delete1({
  ids: ["example-id-1", "example-id-2"],
  namespace: "example-namespace",
});
```

Index.describeIndexStats()

`index.describeIndexStats(requestParameters: DescribeIndexStatsOperationRequest)`

Returns statistics about the index's contents, including the vector count per namespace and the number of dimensions.

Parameters	Type	Description
`requestParameters`	DescribeIndexStatsOperationRequest	Describe index stats request wrapper

Types

DescribeIndexStatsOperationRequest

Parameters	Type	Description
`describeIndexStatsRequest`	DescribeIndexStatsRequest	Describe index stats request parameters

DescribeIndexStatsRequest

parameter	Type	Description
`filter`	object	(Optional) A metadata filter expression.

Returns:

- `namespaces` : `object` A mapping for each namespace in the index from the namespace name to a summary of its contents. If a metadata filter expression is present, the summary will reflect only vectors matching that expression.
- `dimension` : `int64` The dimension of the indexed vectors.
- `indexFullness` : `float` The fullness of the index, regardless of whether a metadata filter expression was passed. The granularity of this metric is 10%.
- `totalVectorCount` : `int64` The total number of vectors in the index.

Example:

JavaScript

```javascript
const indexStats = await index.describeIndexStats({
  describeIndexStatsRequest: {},
});
```

Read more about filtering for more detail.

Index.fetch()

`index.fetch(requestParameters: FetchRequest)`

The Fetch operation looks up and returns vectors, by ID, from a single namespace. The returned vectors include the vector data and metadata.

Parameters	Type	Description
`requestParameters`	FetchRequest	Fetch request parameters

Types

FetchRequest

Parameters	Type	Description
`ids`	Array	The vector IDs to fetch. Does not accept values containing spaces.
`namespace`	string	(Optional) The namespace containing the vectors.

Returns:

- `vectors` : `object` Contains the vectors.
- `namespace` : `string` The namespace of the vectors.

Example:

JavaScript

```javascript
const fetchResponse = await index.fetch({
  ids: ["example-id-1", "example-id-2"],
  namespace: "example-namespace",
});
```

Index.query()

`index.query(requestParameters: QueryOperationRequest)`

Search a namespace using a query vector. Retrieves the ids of the most similar items in a namespace, along with their similarity scores.

Parameters	Type	Description
`requestParameters`	QueryOperationRequest	The query operation request wrapper.

Types

Parameters	Type	Description
`queryRequest`	QueryRequest	The query operation request.

QueryRequest

Parameter	Type	Description
`namespace`	string	(Optional) The namespace to query.

Parameter	Type	Description
`topK`	number	The number of results to return for each query.
`filter`	object	(Optional) The filter to apply. You can use vector metadata to limit your search. See https://www.pinecone.io/docs/metadata-filtering/.
`includeValues`	boolean	(Optional) Indicates whether vector values are included in the response. Defaults to `false`.
`includeMetadata`	boolean	(Optional) Indicates whether metadata is included in the response as well as the ids. Defaults to `false`.
`vector`	Array	(Optional) The query vector. This should be the same length as the dimension of the index being queried. Each `query()` request can contain only one of the parameters `id` or `vector`.
`id`	string	(Optional) The unique ID of the vector to be used as a query vector. Each `query()` request can contain only one of the parameters `vector` or `id`.

Example:

JavaScript

```
const queryResponse = await index.query({
  queryRequest: {
    namespace: "example-namespace",
    topK: 10,
    filter: {
      genre: { $in: ["comedy", "documentary", "drama"] },
    },
    includeValues: true,
    includeMetadata: true,
    vector: [0.1, 0.2, 0.3, 0.4],
  },
});
```

Index.update()

`index.update(requestParameters: UpdateOperationRequest)`

Updates vectors in a namespace. If a value is included, it will overwrite the previous value. If `setMetadata` is included in the `updateRequest`, the values of the fields specified in it will be added or overwrite the previous value.

Parameters	Type	Description
`requestParameters`	UpdateOperationRequest	The update operation wrapper

Types

UpdateOperationRequest

Parameters	Type	Description
`updateRequest`	UpdateRequest	The update request.

UpdateRequest

Parameter	Type	Description
`id`	string	The vector's unique ID.
`values`	Array	(Optional) Vector data.
`setMetadata`	object	(Optional) Metadata to set for the vector.
`namespace`	string	(Optional) The namespace containing the vector.

Example:

JavaScript

```
const updateResponse = await index.update({
  updatedRequest: {
    id: "vec1",
    values: [0.1, 0.2, 0.3, 0.4],
    setMetadata: {
      genre: "drama",
    },
    namespace: "example-namespace",
  },
});
```

Index.upsert()

`index.upsert(requestParameters: UpsertOperationRequest)`

Writes vectors into a namespace. If a new value is upserted for an existing vector ID, it will overwrite the previous value.

Parameters	Type	Description
`requestParameters`	UpsertOperationRequest	Upsert operation wrapper

Types

UpsertOperationRequest

Parameters	Type	Description
`upsertRequest`	UpsertRequest	The upsert request.

UpsertRequest

| Parameter | Type | Description |
| `vectors` | Array | An array containing the vectors to upsert. Recommended batch limit is 100 vectors.
`id` (str) - The vector's unique id.
`values` ([float]) - The vector data.
`metadata` (object) - (Optional) Metadata for the vector. |
| `namespace` | string | (Optional) The namespace name to upsert vectors. |

Vector

Parameter	Type	Description
`id`	string	The vector's unique ID.
`values`	Array	Vector data.
`metadata`	object	(Optional) Metadata for the vector.

Returns:

- `upsertedCount` : `int64` The number of vectors upserted.

Example:

JavaScript

```
const upsertResponse = await index.upsert({
  upsertRequest: {
    vectors: [
      {
        id: "vec1",
        values: [0.1, 0.2, 0.3, 0.4],
        metadata: {
          genre: "drama",
        },
      },
      {
        id: "vec2",
        values: [0.1, 0.2, 0.3, 0.4],
        metadata: {
          genre: "comedy",
        },
      },
    ],
    namespace: "example-namespace",
  },
});
```

Updated5 days ago

Published in Towards Data Science

Sophia Yang

Apr 8 · 6 min read · ✦ Member-only · Listen

4 Ways to Do Question Answering in LangChain

Chat with your long PDF docs: load_qa_chain, RetrievalQA, VectorstoreIndexCreator, ConversationalRetrievalChain

Are you interested in chatting with your own documents, whether it is a text file, a PDF, or a website? LangChain makes it easy for you to do question answering with your documents. But do you know that there are at least 4 ways to do question answering in LangChain? In this blog post, we are going to explore four different ways to do question-answering and the various options you could consider for your use cases.

Before we dive into question answering, you may wonder: what is LangChain? Great question! In my opinion, LangChain is the easiest way to interact with language models and build applications. It is an **open-source** tool that wraps around many LLMs and tools. Check out my previous blog post and video on an overview of LangChain.

Okay, now let's get started with question-answering on external documents.

Code: Check out the code for this blog post here.

Set up OpenAI API

Create an account at OpenAI and create an API key: https://platform.openai.com/account. Note that OpenAI API is not free. You will need to set up billing information there to be able to use OpenAI API. Alternatively, you can use models from HuggingFace Hub or other places. Check out my previous blog post and video on how to use other models.

```
import os
os.environ["OPENAI_API_KEY"] = "COPY AND PASTE YOUR API KEY HERE"
```

Sophia Yang

3.4K Followers

Ph.D. | Senior Data Scientist@Anaconda | Twitter: twitter.com/sophiamyang | YouTube: youtube.com/SophiaYangDS | Book Club: dsbookclub.github.io

Follow

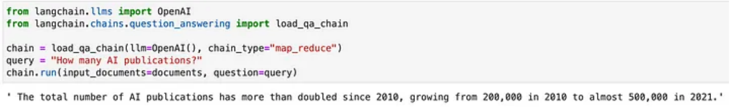
Load documents

LangChain supports many many Document Loaders such as Notion, YouTube, and Figma. In this example, I'd like to chat with my PDF file. Thus, I used the PyPDFLoader to load my file. I'm actually using Chapter 1 of the AI index report, which includes 55 pages, and I saved it in the materials directory of my Github repo.

```
# load document
from langchain.document_loaders import PyPDFLoader
loader = PyPDFLoader("materials/example.pdf")
documents = loader.load()
```

Method 1: load_qa_chain

`load_qa_chain` provides the most generic interface for answering questions. It loads a chain that you can do QA for your input documents and uses ALL of the text in the documents.



```
from langchain.llms import OpenAI
from langchain.chains.question_answering import load_qa_chain

chain = load_qa_chain(llm=OpenAI(), chain_type="map_reduce")
query = "How many AI publications?"
chain.run(input_documents=documents, question=query)
```

```
' The total number of AI publications has more than doubled since 2010, growing from 200,000 in 2010 to almost 500,000 in 2021.'
```

It also lets you do QA over a set of documents:

```
### For multiple documents
loaders = [....]
documents = []
for loader in loaders:
    documents.extend(loader.load())
```

🤔 *But what if my document is super long that it exceeds the token limit?*

There are two ways to fix it:

Solution 1: Chain Type

The default `chain_type="stuff"` uses ALL of the text from the documents in the prompt. It actually doesn't work with our example because it exceeds the token limit and causes rate-limiting errors. That's why in this example, we had to use other chain types for example `"map_reduce"`. What are the other chain types?

- `map_reduce`: It separates texts into batches (as an example, you can define batch size in `llm=OpenAI(batch_size=5)`), feeds each batch with the question to LLM separately, and comes up with the final answer based on the answers from each batch.
- `refine` : It separates texts into batches, feeds the first batch to LLM, and feeds the answer and the second batch to LLM. It refines the answer by going through all the batches.
- `map-rerank`: It separates texts into batches, feeds each batch to LLM, returns a score of how fully it answers the question, and comes up with the final answer based on the high-scored answers from each batch.

Solution 2: RetrievalQA

One issue with using ALL of the text is that it can be very costly because you are feeding all the texts to OpenAI API and the API is charged by the number of tokens. A better solution is to retrieve relevant text chunks first and only use the relevant text chunks in the language model. I'm going to go through the details of RetrievalQA next.

Method 2: RetrievalQA

`RetrievalQA` chain actually uses `load_qa_chain` under the hood. We retrieve the most relevant chunk of text and feed those to the language model.

Here is how it works:

```
from langchain.chains import RetrievalQA
from langchain.indexes import VectorstoreIndexCreator
from langchain.text_splitter import CharacterTextSplitter
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Chroma

# split the documents into chunks
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
texts = text_splitter.split_documents(documents)
# select which embeddings we want to use
embeddings = OpenAIEmbeddings()
# create the vectorestore to use as the index
db = Chroma.from_documents(texts, embeddings)
# expose this index in a retriever interface
retriever = db.as_retriever(search_type="similarity", search_kwargs={"k":2})
# create a chain to answer questions
qa = RetrievalQA.from_chain_type(
    llm=OpenAI(), chain_type="stuff", retriever=retriever, return_source_documents=T
query = "How many AI publications in 2021?"
result = qa({"query": query})
```

In the result, we can see the answer and two source documents because we defined k as 2 meaning that we are only interested in getting two relevant text chunks.

Options:

There are various options for you to choose from in this process:

- embeddings: In the example, we used OpenAI Embeddings. But there are many other embedding options such as Cohere Embeddings, and HuggingFaceEmbeddings from specific models.
- TextSplitter: We used Character Text Splitter in the example where the text is split by a single character. You can also different text splitters and different tokens mentioned in this doc.
- VectorStore: We used Chroma as our vector database where we store our embedded text vectors. Other popular options are FAISS, Mulvus,

and Pinecone.

- Retrievers: We used a VectoreStoreRetriver, which is backed by a VectorStore. To retrieve text, there are two search types you can choose: search_type: “similarity” or “mmr”. `search_type="similarity"` uses similarity search in the retriever object where it selects text chunk vectors that are most similar to the question vector. `search_type="mmr"` uses the maximum marginal relevance search where it optimizes for similarity to query AND diversity among selected documents.
- Chain Type: same as method 1. You can also define the chain type as one of the four options: “stuff”, “map reduce”, “refine”, “map_rerank”.

Method 3: VectorstoreIndexCreator

VectorstoreIndexCreator is a wrapper around the above functionality. It is exactly the same under the hood, but just exposes a higher-level interface to let you get started in three lines of code:

Of course, you can also specify different options in this wrapper:

Method 4: ConversationalRetrievalChain

ConversationalRetrievalChain is very similar to method 2 RetrievalQA. It added an additional parameter `chat_history` to pass in chat history which can be used for follow-up questions.

ConversationalRetrievalChain = conversation memory + RetrievalQAChain

If you would like your language model to have a memory of the previous conversation, use this method. In my example below, I asked about the number of AI publications and got the result of 500,000. Then I asked the LLM to divide this number by 2. Since it has all the chat history, the model knows the number I was referring to is 500,000 and the result returned is 250,000.

Conclusion

Now you know four ways to do question answering with LLMs in LangChain. In summary, load_qa_chain uses all texts and accepts multiple documents; RetrievalQA uses load_qa_chain under the hood but retrieves relevant text chunks first; VectorstoreIndexCreator is the same as RetrievalQA with a higher-level interface; ConversationalRetrievalChain is useful when you want to pass in your chat history to the model.

Acknowledgment:

Thank you Harrison Chase for the guidance!

Photo by FLY:D on Unsplash

. . .

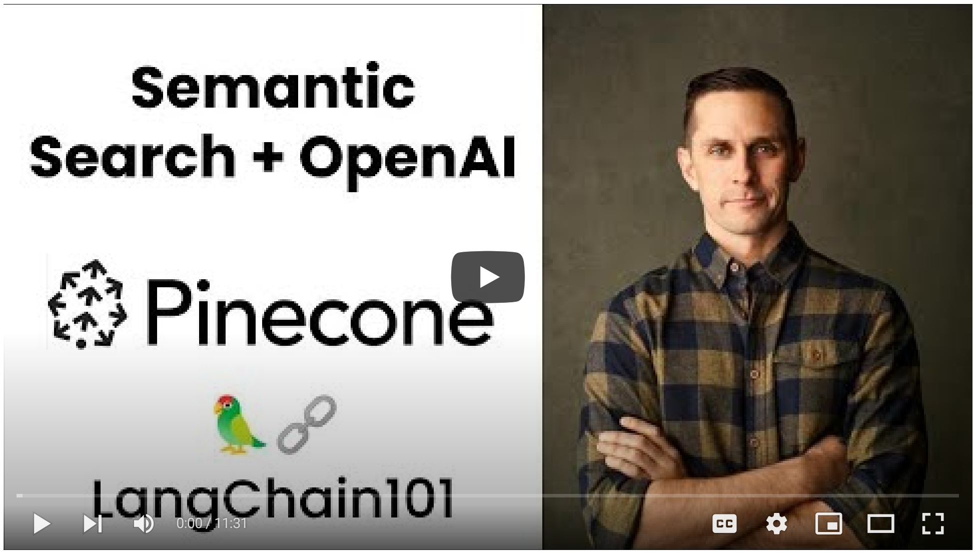
By Sophia Yang on April 8, 2023

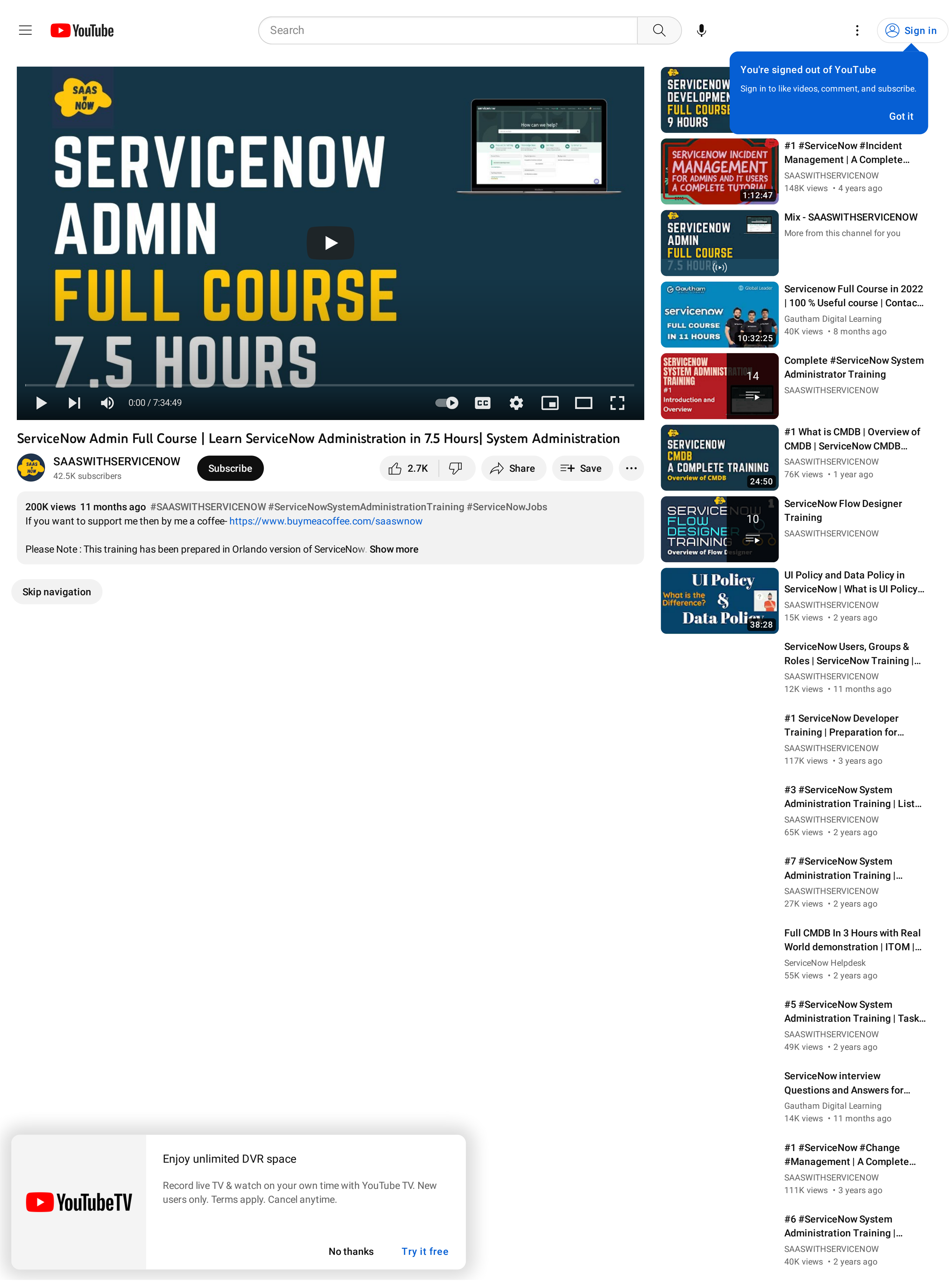
Sophia Yang is a Senior Data Scientist. Connect with me on LinkedIn, Twitter, and YouTube and join the DS/ML Book Club ❤️

Langchain OpenAI Chatbots

-- 3

Search





You're signed out of YouTube

Sign in to like videos, comment, and subscribe.

Got it

ServiceNow Admin Full Course | Learn ServiceNow Administration in 7.5 Hours| System Administration

SAASWITHSERVICENOW

42.5K subscribers

Subscribe

200K views 11 months ago #SAASWITHSERVICENOW #ServiceNowSystemAdministrationTraining #ServiceNowJobs

If you want to support me then by me a coffee- https://www.buymeacoffee.com/saaswnow

Please Note : This training has been prepared in Orlando version of ServiceNow. Show more

Skip navigation

SERVICENOW INCIDENT MANAGEMENT FOR ADMINS AND IT USERS A COMPLETE TUTORIAL
1:12:47

#1 #ServiceNow #Incident Management | A Complete...

SAASWITHSERVICENOW

148K views • 4 years ago

Mix - SAASWITHSERVICENOW

More from this channel for you

Servicenow Full Course in 2022 | 100 % Useful course | Contac...

Gautham Digital Learning

40K views • 8 months ago

SERVICENOW SYSTEM ADMINISTRATION TRAINING
#1 Introduction and Overview
14

Complete #ServiceNow System Administrator Training

SAASWITHSERVICENOW

#1 What is CMDB | Overview of CMDB | ServiceNow CMDB...

SAASWITHSERVICENOW

76K views • 1 year ago

ServiceNow Flow Designer Training

SAASWITHSERVICENOW

UI Policy and Data Policy in ServiceNow | What is UI Policy...

SAASWITHSERVICENOW

15K views • 2 years ago

ServiceNow Users, Groups & Roles | ServiceNow Training |...

SAASWITHSERVICENOW

12K views • 11 months ago

#1 ServiceNow Developer Training | Preparation for...

SAASWITHSERVICENOW

117K views • 3 years ago

#3 #ServiceNow System Administration Training | List...

SAASWITHSERVICENOW

65K views • 2 years ago

#7 #ServiceNow System Administration Training |...

SAASWITHSERVICENOW

27K views • 2 years ago

Full CMDB In 3 Hours with Real World demonstration | ITOM |...

ServiceNow Helpdesk

55K views • 2 years ago

#5 #ServiceNow System Administration Training | Task...

SAASWITHSERVICENOW

49K views • 2 years ago

ServiceNow interview Questions and Answers for...

Gautham Digital Learning

14K views • 11 months ago

#1 #ServiceNow #Change #Management | A Complete...

SAASWITHSERVICENOW

111K views • 3 years ago

#6 #ServiceNow System Administration Training |...

SAASWITHSERVICENOW

40K views • 2 years ago

Search

Sign in

You're signed out of YouTube

Sign in to like videos, comment, and subscribe.

Got it

48K views • 2 years ago

ServiceNow most asked 35 Interview questions with...

SKFacts and ITCareers

34K views • 11 months ago

Task Requirements:

Step Number 2:

```
from langchain.document_loaders import UnstructuredPDFLoader, OnlinePDFLoader

from langchain.text_splitter import RecursiveCharacterTextSplitter
```

Load your data

In [2]:

```
loader = UnstructuredPDFLoader("../data/field-guide-to-data-science.pdf")

# loader = OnlinePDFLoader("https://wolfpaulus.com/wp-content/uploads/2017/05/field-guide-to-data-science.pdf")
```

In [3]:

```
data = loader.load()
```

In [4]:

```
print (f'You have {len(data)} document(s) in your data')

print (f'There are {len(data[0].page_content)} characters in your document')
```

```
You have 1 document(s) in your data

There are 176584 characters in your document
```

Chunk your data up into smaller documents

In [5]:

Task Requirements:

```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=0)

texts = text_splitter.split_documents(data)
```

In [6]:

```
print (f'Now you have {len(texts)} documents')
```

```
Now you have 228 documents
```

Create embeddings of your documents to get ready for semantic search

In [9]:

```
from langchain.vectorstores import Chroma, Pinecone

from langchain.embeddings.openai import OpenAIEmbeddings

import pinecone
```

In [24]:

```
OPENAI_API_KEY = '...'

PINECONE_API_KEY = '...'

PINECONE_API_ENV = 'us-east1-gcp'
```

In [11]:

```
embeddings = OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY)
```

In [12]:

```
# initialize pinecone
```

Task Requirements:

```
pinecone.init(

    api_key=PINECONE_API_KEY,  # find at app.pinecone.io

    environment=PINECONE_API_ENV  # next to api key in console

)

index_name = "langchain2"
```

In [13]:

```
docsearch = Pinecone.from_texts([t.page_content for t in texts], embeddings,
index_name=index_name)
```

In [14]:

```
query = "What are examples of good data science teams?"

docs = docsearch.similarity_search(query, include_metadata=True)
```

Query those docs to get your answer back

In [16]:

```
from langchain.llms import OpenAI

from langchain.chains.question_answering import load_qa_chain
```

In [17]:

```
llm = OpenAI(temperature=0, openai_api_key=OPENAI_API_KEY)

chain = load_qa_chain(llm, chain_type="stuff")
```

In [22]:

Task Requirements:

```
query = "What is the collect stage of data maturity?"


docs = docsearch.similarity_search(query, include_metadata=True)
```

In [23]:

```
chain.run(input_documents=docs, question=query)
```

Out[23]:

```
' The collect stage of data maturity focuses on collecting internal or external datasets. Examples include gathering sales records and corresponding weather data.'
```

In [ ]:

LangChain QA

All code comes from LangChain docs.

In [ ]:

```
!pip install langchain openai chromadb tiktoken pypdf
```

In [ ]:

```
import os
os.environ["OPENAI_API_KEY"] = ""
```

In [ ]:

```
from langchain.chains import RetrievalQA
from langchain.llms import OpenAI
from langchain.document_loaders import TextLoader
from langchain.document_loaders import PyPDFLoader
from langchain.indexes import VectorstoreIndexCreator
from langchain.text_splitter import CharacterTextSplitter
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Chroma
```

In [ ]:

```
llm = OpenAI()
print(llm("tell me a joke"))
```

Task Requirements:

load_qa_chain

Loads a chain that you can use to do QA over a set of documents, but it uses ALL of those documents.

chain_type="stuff" will not work because the number of tokens exceeds the limit. We can try other chain types like "map_reduce".

In [ ]:

```
from langchain.chains.question_answering import load_qa_chain

# load document
loader = PyPDFLoader("materials/example.pdf")
documents = loader.load()

### For multiple documents
# loaders = [....]
# documents = []
# for loader in loaders:
#     documents.extend(loader.load())

chain = load_qa_chain(llm=OpenAI(), chain_type="map_reduce")
query = "what is the total number of AI publications?"
chain.run(input_documents=documents, question=query)
```

RetrievalQA

RetrievalQA chain uses load_qa_chain under the hood. We retrieve the most relevant chunck of text and feed those to the language model.

Options:

- embeddings
- TextSplitter
- VectorStore
- Retrievers
    - search_type: "similarity" or "mmr"
- Chain Type: "stuff", "map reduce", "refine", "map_rerank"

In [ ]:

```
# load document
loader = PyPDFLoader("materials/example.pdf")
documents = loader.load()
# split the documents into chunks
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
texts = text_splitter.split_documents(documents)
# select which embeddings we want to use
embeddings = OpenAIEmbeddings()
# create the vectorestore to use as the index
```

Task Requirements:

```
db = Chroma.from_documents(texts, embeddings)
# expose this index in a retriever interface
retriever = db.as_retriever(search_type="similarity", search_kwargs={"k":2})
# create a chain to answer questions
qa = RetrievalQA.from_chain_type(
    llm=OpenAI(), chain_type="stuff", retriever=retriever, return_source_documents=True)
query = "what is the total number of AI publications?"
result = qa({"query": query})
```

In [ ]:

```
retriever.get_relevant_documents(query)
```

In [ ]:

```
result
```

VectorstoreIndexCreator

VectorstoreIndexCreator is a wrapper for the above logic.

Source:

- https://python.langchain.com/en/latest/modules/chains/getting_started.html
- https://github.com/hwchase17/langchain/blob/master/langchain/indexes/vectorstore.py#L21-L74

In [ ]:

```
index = VectorstoreIndexCreator(
    # split the documents into chunks
    text_splitter=CharacterTextSplitter(chunk_size=1000, chunk_overlap=0),
    # select which embeddings we want to use
    embedding=OpenAIEmbeddings(),
    # use Chroma as the vectorestore to index and search embeddings
    vectorstore_cls=Chroma
).from_loaders([loader])
query = "what is the total number of AI publications?"
index.query(llm=OpenAI(), question=query, chain_type="stuff")
```

ConversationalRetrievalChain

conversation memory + RetrievalQAChain

Allow for passing in chat history which can be used for follow up questions.

Source:
https://python.langchain.com/en/latest/modules/chains/index_examples/chat_vector_db.html

Task Requirements:

In [ ]:

```
from langchain.chains import ConversationalRetrievalChain
```

In [ ]:

```
# load document
loader = PyPDFLoader("materials/example.pdf")
documents = loader.load()
# split the documents into chunks
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
texts = text_splitter.split_documents(documents)
# select which embeddings we want to use
embeddings = OpenAIEmbeddings()
# create the vectorestore to use as the index
db = Chroma.from_documents(texts, embeddings)
# expose this index in a retriever interface
retriever = db.as_retriever(search_type="similarity", search_kwargs={"k":2})
# create a chain to answer questions
qa = ConversationalRetrievalChain.from_llm(OpenAI(), retriever)
chat_history = []
query = "what is the total number of AI publications?"
result = qa({"question": query, "chat_history": chat_history})
```

In [ ]:

```
chat_history = []
query = "what is the total number of AI publications?"
result = qa({"question": query, "chat_history": chat_history})
```

In [ ]:

```
result["answer"]
```

In [ ]:

```
chat_history = [(query, result["answer"])]
query = "What is this number divided by 2?"
result = qa({"question": query, "chat_history": chat_history})
```

In [ ]:

```
chat_history
```

In [ ]:

```
 result['answer']
```

Step Number 3:

A- PineCone package

Task Requirements:

This page provides installation instructions, usage examples, and a reference for the Pinecone Node.JS client.

⚠️
Warning

This is a public preview ("Beta") client. Test thoroughly
before
using this client for production workloads. No SLAs or
technical support
commitments are provided for this client. Expect potential
breaking
changes in future releases.

Getting Started
Installation
Use the following shell command to install the Node.JS client for use with Node.JS versions 17 and above:

Shell

```
npm install @pinecone-database/pinecone
```

Alternatively, you can install Pinecone with Yarn:

Shell

```
yarn add @pinecone-database/pinecone
```

Usage
Initialize the client
To initialize the client, instantiate the PineconeClient class and call the init method. The init method takes an object with the apiKey and environment properties:

JavaScript

```
import { PineconeClient } from
"@pinecone-database/pinecone";

const pinecone = new PineconeClient();
await pinecone.init({
  environment: "YOUR_ENVIRONMENT",
  apiKey: "YOUR_API_KEY",
});
```

Create index

The following example creates an index without a metadata configuration. By default, Pinecone indexes all metadata.

JavaScript

```
await pinecone.createIndex({
  createRequest: {
    name: "example-index",
    dimension: 1024,
  },
});
```

The following example creates an index that only indexes the "color" metadata field. Queries against this index cannot filter based on any other metadata field.

JavaScript

```
await pinecone.createIndex({
  createRequest: {
    name: "example-index-2",
    dimension: 1024,
    metadata_config: {
      indexed: ["color"],
    },
  },
});
```

List indexes

The following example logs all indexes in your project.

JavaScript

```
const indexesList = await pinecone.listIndexes();
```

Describe index

The following example logs information about the index example-index.

Task Requirements:

JavaScript

```
const indexDescription = await pinecone.describeIndex({
  indexName: "example-index",
});
```

Delete index

The following example deletes example-index.

JavaScript

```
await pinecone.deleteIndex({
  indexName: "example-index",
});
```

Scale replicas

The following example sets the number of replicas and pod type for example-index.

JavaScript

```
await pinecone.configureIndex({
  indexName: "example-index",
  patchRequest: {
    replicas: 2,
    podType: "p2",
  },
});
```

Describe index statistics

The following example returns statistics about the index example-index.

JavaScript

```
const index = pinecone.Index("example-index");
const indexStats = index.describeIndexStats({
  describeIndexStatsRequest: {
    filter: {},
  },
});
```

Upsert vectors

The following example upserts vectors to example-index.

Task Requirements:

JavaScript

```
const index = pinecone.Index("example-index");
const upsertRequest = {
  vectors: [
    {
      id: "vec1",
      values: [0.1, 0.2, 0.3, 0.4],
      metadata: {
        genre: "drama",
      },
    },
    {
      id: "vec2",
      values: [0.2, 0.3, 0.4, 0.5],
      metadata: {
        genre: "action",
      },
    },
  ],
  namespace: "example-namespace",
};
const upsertResponse = await index.upsert({ upsertRequest
});
```

Query an index

The following example queries the index example-index with metadata filtering.

JavaScript

```
const index = pinecone.Index("example-index");
const queryRequest = {
  vector: [0.1, 0.2, 0.3, 0.4],
  topK: 10,
  includeValues: true,
  includeMetadata: true,
  filter: {
    genre: { $in: ["comedy", "documentary", "drama"] },
  },
```

Task Requirements:

```
  namespace: "example-namespace",
};
const queryResponse = await index.query({ queryRequest });
```

Delete vectors

The following example deletes vectors by ID.

JavaScript

```
const index = pinecone.Index("example-index");
await index.delete1({
  ids: ["vec1", "vec2"],
  namespace: "example-namespace",
});
```

Fetch vectors

The following example fetches vectors by ID.

JavaScript

```
const index = pinecone.Index("example-index");
const fetchResponse = await index.fetch({
  ids: ["vec1", "vec2"],
  namespace: "example-namespace",
});
```

Update vectors

The following example updates vectors by ID.

JavaScript

```
const index = pinecone.Index("example-index");
const updateRequest = {
  id: "vec1",
  values: [0.1, 0.2, 0.3, 0.4],
  setMetadata: { genre: "drama" },
  namespace: "example-namespace",
};
const updateResponse = await index.update({
updateRequest });
```

Create collection

The following example creates the collection example-collection from example-index.

Task Requirements:

JavaScript

```
const createCollectionRequest = {
  name: "example-collection",
  source: "example-index",
};

await pinecone.createCollection({
  createCollectionRequest,
});
```

List collections

The following example returns a list of the collections in the current project.

JavaScript

```
const collectionsList = await pinecone.listCollections();
```

Describe a collection

The following example returns a description of the collection example-collection.

JavaScript

```
const collectionDescription = await
pinecone.describeCollection({
  collectionName: "example-collection",
});
```

Delete a collection

The following example deletes the collection example-collection.

JavaScript

```
await pinecone.deleteCollection({
  collectionName: "example-collection",
});
```

Reference

For the REST API or other clients, see the API reference.

Task Requirements:

init()
pinecone.init(configuration: PineconeClientConfiguration)

Initialize the Pinecone client.

Parameters	Type	Description
configuration	PineconeClientConfiguration	The configuration for the Pinecone client.

Types

PineconeClientConfiguration

Parameters	Type	Description
apiKey	string	The API key for the Pinecone service.
environment	string	The cloud environment of your Pinecone project.

Example:

JavaScript

```
import { PineconeClient } from
"@pinecone-database/pinecone";
const pinecone = new PineconeClient();
await pinecone.init({
  apiKey: "YOUR_API_KEY",
  environment: "YOUR_ENVIRONMENT",
});
```

configureIndex()
pinecone.configure_index(indexName: string, patchRequest?: PatchRequest)

Configure an index to change pod type and number of replicas.

Parameters	Type	Description
requestParameters	ConfigureIndexRequest	Index configuration parameters.

Types

ConfigureIndexRequest

Parameters	Type	Description
indexName	string	The name of the index.

Task Requirements:

patchRequest PatchRequest (Optional) Patch request parameters.

PatchRequest

Parameters Type Description

replicas number (Optional) The number of replicas to configure for this index.

podType string (Optional) The new pod type for the index. One of s1, p1, or p2 appended with . and one of x1, x2, x4, or x8.

Example:

JavaScript

```
const newNumberOfReplicas = 4;
const newPodType = "s1.x4";
await pinecone.configureIndex({
  indexName: "example-index",
  patchRequest: {
    replicas: newNumberOfReplicas,
    podType: newPodType,
  },
});
```

createCollection()

pinecone.createCollection(requestParameters: CreateCollectionOperationRequest)

Create a collection from an index.

Parameters Type Description

requestParameters CreateCollectionOperationRequest Create collection operation wrapper

Types

CreateCollectionOperationRequest

Parameters Type Description

createCollectionRequest CreateCollectionRequest Collection request parameters.

CreateCollectionRequest

Parameters Type Description

name string The name of the collection to be created.

Task Requirements:

source string The name of the source index to be used as the source for the collection.

Example:

JavaScript

```
await pinecone.createCollection({
  createCollectionRequest: {
    name: "example-collection",
    source: "example-index",
  },
});
```

createIndex()

pinecone.createIndex(requestParameters?: CreateIndexRequest)

Create an index.

Parameters	Type	Description
requestParameters	CreateIndexRequest	Create index operation wrapper

Types

CreateIndexRequest

Parameters	Type	Description
createRequest	CreateRequest	Create index request parameters

CreateRequest

Parameters	Type	Description
name	str	The name of the index to be created. The maximum length is 45 characters.
dimension	integer	The dimensions of the vectors to be inserted in the index.
metric	str	(Optional) The distance metric to be used for similarity search: 'euclidean', 'cosine', or 'dotproduct'.
pods	int	(Optional) The number of pods for the index to use, including replicas.
replicas	int	(Optional) The number of replicas.
pod_type	str	(Optional) The new pod type for the index. One of s1, p1, or p2 appended with . and one of x1, x2, x4, or x8.

Task Requirements:

metadata_config object (Optional) Configuration for the behavior of Pinecone's internal metadata index. By default, all metadata is indexed; when metadata_config is present, only specified metadata fields are indexed. To specify metadata fields to index, provide a JSON object of the following form: {"indexed": ["example_metadata_field"]}
source_collection str (Optional) The name of the collection to create an index from.
Example:

JavaScript

```
// The following example creates an index without a metadata
// configuration. By default, Pinecone indexes all metadata.
await pinecone.createIndex({
  createRequest: {
    name: "pinecone-index",
    dimension: 1024,
  },
});

// The following example creates an index that only indexes
// the 'color' metadata field. Queries against this index
// cannot filter based on any other metadata field.

await pinecone.createIndex({
  createRequest: {
    name: "example-index-2",
    dimension: 1024,
    metadata_config: {
      indexed: ["color"],
    },
  },
});
deleteCollection()
pinecone.deleteCollection(requestParameters:
DeleteCollectionRequest)
```

Delete an existing collection.

Task Requirements:

Types

Parameters	Type	Description
requestParameters	DeleteCollectionRequest	Delete collection request parameters

DeleteCollectionRequest

Parameters	Type	Description
collectionName	string	The name of the collection to delete.

Example:

JavaScript

```
await pinecone.deleteCollection({
  collectionName: "example-collection",
});
```

deleteIndex()

pinecone.deleteIndex(requestParameters: DeleteIndexRequest)

Delete an index.

Types

Parameters	Type	Description
requestParameters	DeleteIndexRequest	Delete index request parameters

DeleteIndexRequest

Parameters	Type	Description
indexName	string	The name of the index to delete.

Example:

JavaScript

```
await pinecone.deleteIndex({
  indexName: "example-index",
});
```

describeCollection()

pinecone.describeCollection(requestParameters: DescribeCollectionRequest)

Get a description of a collection.

Task Requirements:

Types

Parameters	Type	Description
requestParameters	DescribeCollectionRequest	Describe collection request parameters

DescribeCollectionRequest

Parameters	Type	Description
collectionName	string	The name of the collection.

Example:

JavaScript

```
const collectionDescription = await
pinecone.describeCollection({
  collectionName: "example-collection",
});
```

Return:

collectionMeta : object Configuration information and deployment status of the collection.
name : string The name of the collection.
size: integer The size of the collection in bytes.
status: string The status of the collection.
describeIndex()
pinecone.describeIndex(requestParameters: DescribeIndexRequest)

Get a description of an index.

Types

Parameters	Type	Description
requestParameters	DescribeIndexRequest	Describe index request parameters

DescribeIndexRequest

Parameters	Type	Description
indexName	string	The name of the index.

Types
Returns:

database : object

Task Requirements:

name : string The name of the index.
dimension : integer The dimensions of the vectors to be inserted in the index.
metric : string The distance metric used for similarity search: 'euclidean', 'cosine', or 'dotproduct'.
pods : integer The number of pods the index uses, including replicas.
replicas : integer The number of replicas.
pod_type : string The pod type for the index. One of s1, p1, or p2 appended with . and one of x1, x2, x4, or x8.
metadata_config: object Configuration for the behavior of Pinecone's internal metadata index. By default, all metadata is indexed; when metadata_config is present, only specified metadata fields are indexed. To specify metadata fields to index, provide a JSON object of the following form:
{"indexed": ["example_metadata_field"]}
status : object
ready : boolean Whether the index is ready to serve queries.
state : string One of Initializing, ScalingUp, ScalingDown, Terminating, or Ready.
Example:

JavaScript

```
const indexDescription = await pinecone.describeIndex({
  indexName: "example-index",
});
```

listCollections
pinecone.listCollections()

Return a list of the collections in your project.

Example:

JavaScript

```
const collections = await pinecone.listCollections();
```

Returns:

array of strings The names of the collections in your project.

Task Requirements:

listIndexes
pinecone.listIndexes()

Return a list of your Pinecone indexes.

Returns:

array of strings The names of the indexes in your project.
Example:

JavaScript

```
const indexesList = await pinecone.listIndexes();
```

Index()
pinecone.Index(indexName: string)

Construct an Index object.

Parameters	Type	Description
indexName	string	The name of the index.

Example:

JavaScript

```
const index = pinecone.Index("example-index");
```

Index.delete1()
index.delete(requestParameters: Delete1Request)

Delete items by their ID from a single namespace.

Parameters	Type	Description
requestParameters	Delete1Request	Delete request parameters

Types
Delete1Request

Parameters	Type	Description
ids	Array	(Optional) The IDs of the items to delete.
deleteAll	boolean	(Optional) Indicates that all vectors in the index namespace should be deleted.

Task Requirements:

namespace str (Optional) The namespace to delete vectors from, if applicable.
Types
Example:

JavaScript

```
await index.delete1({
  ids: ["example-id-1", "example-id-2"],
  namespace: "example-namespace",
});
```

Index.describeIndexStats()
index.describeIndexStats(requestParameters: DescribeIndexStatsOperationRequest)

Returns statistics about the index's contents, including the vector count per namespace and the number of dimensions.

Parameters	Type	Description
requestParameters	DescribeIndexStatsOperationRequest	Describe index stats request wrapper

Types
DescribeIndexStatsOperationRequest

Parameters	Type	Description
describeIndexStatsRequest	DescribeIndexStatsRequest	Describe index stats request parameters

DescribeIndexStatsRequest

parameter	Type	Description
filter	object	(Optional) A metadata filter expression.

Returns:

namespaces : object A mapping for each namespace in the index from the namespace name to a
summary of its contents. If a metadata filter expression is present, the summary will reflect only vectors matching that expression.
dimension : int64 The dimension of the indexed vectors.
indexFullness : float The fullness of the index, regardless of whether a metadata filter expression was passed. The granularity of this metric is 10%.

Task Requirements:

totalVectorCount : int64 The total number of vectors in the index.
Example:

JavaScript

```
const indexStats = await index.describeIndexStats({
  describeIndexStatsRequest: {},
});
```

Read more about filtering for more detail.

Index.fetch()
index.fetch(requestParameters: FetchRequest)

The Fetch operation looks up and returns vectors, by ID, from a single namespace. The returned vectors include the vector data and metadata.

Parameters	Type	Description
requestParameters	FetchRequest	Fetch request parameters

Types
FetchRequest

Parameters	Type	Description
ids	Array	The vector IDs to fetch. Does not accept values containing spaces.
namespace	string	(Optional) The namespace containing the vectors.

Returns:

vectors : object Contains the vectors.
namespace : string The namespace of the vectors.
Example:

JavaScript

```
const fetchResponse = await index.fetch({
  ids: ["example-id-1", "example-id-2"],
  namespace: "example-namespace",
});
```

Task Requirements:

Index.query()
index.query(requestParameters: QueryOperationRequest)

Search a namespace using a query vector. Retrieves the ids of the most similar items in a namespace, along with their similarity scores.

Parameters	Type	Description
requestParameters	QueryOperationRequest	The query operation request wrapper.

Types

Parameters	Type	Description
queryRequest	QueryRequest	The query operation request.

QueryRequest

Parameter	Type	Description
namespace	string	(Optional) The namespace to query.
topK	number	The number of results to return for each query.
filter	object	(Optional) The filter to apply. You can use vector metadata to limit your search. See https://www.pinecone.io/docs/metadata-filtering/.
includeValues	boolean	(Optional) Indicates whether vector values are included in the response. Defaults to false.
includeMetadata	boolean	(Optional) Indicates whether metadata is included in the response as well as the ids. Defaults to false.
vector	Array	(Optional) The query vector. This should be the same length as the dimension of the index being queried. Each query() request can contain only one of the parameters id or vector.
id	string	(Optional) The unique ID of the vector to be used as a query vector. Each query() request can contain only one of the parameters vector or id.

Example:

JavaScript

```
const queryResponse = await index.query({
  queryRequest: {
    namespace: "example-namespace",
    topK: 10,
    filter: {
      genre: { $in: ["comedy", "documentary", "drama"] },
    },
    includeValues: true,
    includeMetadata: true,
    vector: [0.1, 0.2, 0.3, 0.4],
  },
});
```

Index.update()

index.update(requestParameters: UpdateOperationRequest)

Updates vectors in a namespace. If a value is included, it will overwrite the previous value.

If setMetadata is included in the updateRequest, the values of the fields specified in it will be added or overwrite the previous value.

Parameters	Type	Description
requestParameters	UpdateOperationRequest	The update operation wrapper

Types

UpdateOperationRequest

Parameters	Type	Description
updateRequest	UpdateRequest	The update request.

UpdateRequest

Parameter	Type	Description
id	string	The vector's unique ID.
values	Array	(Optional) Vector data.
setMetadata	object	(Optional) Metadata to set for the vector.
namespace	string	(Optional) The namespace containing the vector.

Example:

JavaScript

```
const updateResponse = await index.update({
  updatedRequest: {
```

Task Requirements:

```
    id: "vec1",
    values: [0.1, 0.2, 0.3, 0.4],
    setMetadata: {
      genre: "drama",
    },
    namespace: "example-namespace",
  },
});
Index.upsert()
index.upsert(requestParameters: UpsertOperationRequest)
```

Writes vectors into a namespace. If a new value is upserted for an existing vector ID, it will overwrite the previous value.

Parameters	Type	Description
requestParameters	UpsertOperationRequest	Upsert operation wrapper

Types

UpsertOperationRequest

Parameters	Type	Description
upsertRequest	UpsertRequest	The upsert request.

UpsertRequest

| Parameter | Type | Description |
| vectors | Array | An array containing the vectors to upsert. Recommended batch limit is 100 vectors.
id (str) - The vector's unique id.
values ([float]) - The vector data.
metadata (object) - (Optional) Metadata for the vector. |
| namespace | string | (Optional) The namespace name to upsert vectors. |

Vector

Parameter	Type	Description
id	string	The vector's unique ID.
values	Array	Vector data.
metadata	object	(Optional) Metadata for the vector.

Returns:

upsertedCount : int64 The number of vectors upserted.

Example:

Task Requirements:

JavaScript

```
const upsertResponse = await index.upsert({
  upsertRequest: {
    vectors: [
      {
        id: "vec1",
        values: [0.1, 0.2, 0.3, 0.4],
        metadata: {
          genre: "drama",
        },
      },
      {
        id: "vec2",
        values: [0.1, 0.2, 0.3, 0.4],
        metadata: {
          genre: "comedy",
        },
      },
    ],
    namespace: "example-namespace",
  },
});
```

b- Open AI

OpenAI Node.js Library

The OpenAI Node.js library provides convenient access to the OpenAI API from Node.js applications. Most of the code in this library is generated from our OpenAPI specification.

Important note: this library is meant for server-side usage only, as using it in client-side browser code will expose your secret API key. See here for more details.

Task Requirements:

Installation

```
$ npm install openai
```

Usage

The library needs to be configured with your account's secret key, which is available on the website. We recommend setting it as an environment variable. Here's an example of initializing the library with the API key loaded from an environment variable and creating a completion:

```
const { Configuration, OpenAIApi } = require("openai");



const configuration = new Configuration({

  apiKey: process.env.OPENAI_API_KEY,

});

const openai = new OpenAIApi(configuration);



const completion = await openai.createCompletion({

  model: "text-davinci-003",

  prompt: "Hello world",

});
```

Task Requirements:

```
console.log(completion.data.choices[0].text);
```

Check out the full API documentation for examples of all the available functions.

Request options

All of the available API request functions additionally contain an optional final parameter where you can pass custom axios request options, for example:

```
const completion = await openai.createCompletion(
  {
    model: "text-davinci-003",
    prompt: "Hello world",
  },
  {
    timeout: 1000,
    headers: {
      "Example-Header": "example",
    },
  }
);
```

Task Requirements:

Error handling

API requests can potentially return errors due to invalid inputs or other issues. These errors can be handled with a `try...catch` statement, and the error details can be found in either `error.response` or `error.message`:

```
try {
  const completion = await openai.createCompletion({
    model: "text-davinci-003",
    prompt: "Hello world",
  });
  console.log(completion.data.choices[0].text);
} catch (error) {
  if (error.response) {
    console.log(error.response.status);
    console.log(error.response.data);
  } else {
    console.log(error.message);
  }
}
```

Streaming completions

Task Requirements:

Streaming completions (`stream=true`) are not natively supported in this package yet, but a workaround exists if needed.

Upgrade guide

All breaking changes for major version releases are listed below.

3.0.0

- The function signature of `createCompletion(engineId, params)` changed to `createCompletion(params)`. The value previously passed in as the `engineId` argument should now be passed in as `model` in the params object (e.g. `createCompletion({ model: "text-davinci-003", ... })`)
- Replace any `createCompletionFromModel(params)` calls with `createCompletion(params)`

Thanks

Thank you to ceifa for creating and maintaining the original unofficial `openai` npm package before we released this official library! ceifa's original package has been renamed to gpt-x.

Keywords

openai
open
ai
gpt-3
Gpt3

Task Requirements:

C-Langchain

Welcome to LangChain

LangChain is a framework for developing applications powered by language models. We believe that the most powerful and differentiated applications will not only call out to a language model via an api, but will also:

- *Be data-aware*: connect a language model to other sources of data
- *Be agentic*: allow a language model to interact with its environment

The LangChain framework is designed with the above principles in mind.

Getting Started

Checkout the below guide for a walkthrough of how to get started using LangChain to create an Language Model application.

- Getting Started Documentation

Components

There are several main modules that LangChain provides support for. For each module we provide some examples to get started and get familiar with some of the concepts. These modules are, in increasing order of complexity:

- Prompts: This includes prompt management, prompt optimization, and prompt serialization.
- LLMs: This includes a generic interface for all LLMs, and common utilities for working with LLMs.
- Indexes: This includes patterns and functionality for structuring your own text data so it can interact with language models (including embeddings, vectorstores, text splitters, retrievers, etc).
- Memory: Memory is the concept of persisting state between calls of a chain/agent. LangChain provides a standard interface for memory, a collection of memory implementations, and examples of chains/agents that use memory.
- Chains: Chains go beyond just a single LLM call, and are sequences of calls (whether to an LLM or a different utility). LangChain provides a standard interface for chains, lots of integrations with other tools, and end-to-end chains for common applications.
- Agents: Agents involve an LLM making decisions about which Actions to take, taking that Action, seeing an Observation, and repeating that until

done. LangChain provides a standard interface for agents, a selection of agents to choose from, and examples of end to end agents.

Reference Docs

All of LangChain's reference documentation, in one place. Full documentation on all methods and classes.

Production

As you move from prototyping into production, we're developing resources to help you do so. These including:

- Deployment: resources on how to deploy your end application.
- Tracing: resouces on how to use tracing to log and debug your applications.

Additional Resources

Additional collection of resources we think may be useful as you develop your application!

- LangChainHub: The LangChainHub is a place to share and explore other prompts, chains, and agents.
- Discord: Join us on our Discord to discuss all things LangChain!
- Production Support: As you move your LangChains into production, we'd love to offer more comprehensive support. Please fill out this form and we'll set up a dedicated support Slack channel