
LLM documentation

Release 0.12-1-gbdadd31

Simon Willison

Nov 12, 2023

CONTENTS

1	Quick start	3
2	Contents	5
2.1	Setup	5
2.1.1	Installation	5
2.1.2	Upgrading to the latest version	5
2.1.3	Installing plugins	6
2.1.4	API key management	6
2.1.5	Configuration	7
2.2	Usage	8
2.2.1	Executing a prompt	8
2.2.2	Completion prompts	9
2.2.3	Continuing a conversation	9
2.2.4	Using with a shell	9
2.2.5	System prompts	10
2.2.6	Starting an interactive chat	10
2.2.7	Listing available models	12
2.3	Other models	14
2.3.1	Installing and using a local model	14
2.3.2	Adding more OpenAI models	14
2.3.3	OpenAI-compatible models	15
2.4	Embeddings	16
2.4.1	Embedding with the CLI	17
2.4.2	Using embeddings from Python	24
2.4.3	Writing plugins to add new embedding models	28
2.4.4	Embedding storage format	29
2.5	Plugins	30
2.5.1	Installing plugins	30
2.5.2	Plugin directory	32
2.5.3	Plugin hooks	33
2.5.4	Writing a plugin to support a new model	34
2.5.5	Utility functions for plugins	46
2.6	Model aliases	47
2.6.1	Listing aliases	48
2.6.2	Adding a new alias	48
2.6.3	Removing an alias	49
2.6.4	Viewing the aliases file	49
2.7	Python API	49
2.7.1	Basic prompt execution	49
2.7.2	Streaming responses	51

2.7.3	Conversations	51
2.7.4	Other functions	51
2.8	Prompt templates	52
2.8.1	Getting started	52
2.8.2	Using a template	52
2.8.3	Listing available templates	53
2.8.4	Templates as YAML files	53
2.8.5	System templates	54
2.8.6	Additional template variables	54
2.8.7	Specifying default parameters	55
2.8.8	Setting a default model for a template	55
2.9	Logging to SQLite	56
2.9.1	Viewing the logs	56
2.9.2	SQL schema	58
2.10	Related tools	58
2.10.1	strip-tags	58
2.10.2	ttok	59
2.10.3	Symbex	59
2.11	CLI reference	59
2.11.1	llm -help	59
2.12	Contributing	72
2.12.1	Debugging tricks	72
2.12.2	Documentation	72
2.12.3	Release process	73
2.13	Changelog	73
2.13.1	0.12 (2023-11-06)	73
2.13.2	0.11.2 (2023-11-06)	73
2.13.3	0.11.1 (2023-10-31)	73
2.13.4	0.11 (2023-09-18)	73
2.13.5	0.10 (2023-09-12)	74
2.13.6	0.10a1 (2023-09-11)	76
2.13.7	0.10a0 (2023-09-04)	76
2.13.8	0.9 (2023-09-03)	77
2.13.9	0.8.1 (2023-08-31)	77
2.13.10	0.8 (2023-08-20)	78
2.13.11	0.7.1 (2023-08-19)	78
2.13.12	0.7 (2023-08-12)	78
2.13.13	0.6.1 (2023-07-24)	79
2.13.14	0.6 (2023-07-18)	79
2.13.15	0.5 (2023-07-12)	79
2.13.16	0.4.1 (2023-06-17)	80
2.13.17	0.4 (2023-06-17)	81
2.13.18	0.3 (2023-05-17)	83
2.13.19	0.2 (2023-04-01)	83
2.13.20	0.1 (2023-04-01)	83

A CLI utility and Python library for interacting with Large Language Models, both via remote APIs and models that can be installed and run on your own machine.

Run prompts from the command-line, store the results in SQLite, generate embeddings and more.

Background on this project:

- llm, tok and strip-tags—CLI tools for working with ChatGPT and other LLMs
- The LLM CLI tool now supports self-hosted language models via plugins
- Accessing Llama 2 from the command-line with the llm-replicate plugin
- Run Llama 2 on your own Mac using LLM and Homebrew
- Catching up on the weird world of LLMs
- LLM now provides tools for working with embeddings
- Build an image search engine with llm-clip, chat with models with llm chat

For more check out [the llm tag](#) on my blog.

QUICK START

First, install LLM using pip:

```
pip install llm
```

Or with `pipx` (recommended, as then it won't clash with any other installed packages):

```
pipx install llm
```

If you have an OpenAI API key you can run this:

```
# Paste your OpenAI API key into this
llm keys set openai

# Run a prompt
llm "Ten fun names for a pet pelican"

# Run a system prompt against a file
cat myfile.py | llm -s "Explain this code"
```

Or you can *install a plugin* and use models that can run on your local device:

```
# Install the plugin
llm install llm-gpt4all

# Download and run a prompt against the Orca Mini 7B model
llm -m orca-mini-3b-gguf2-q4_0 'What is the capital of France?'
```

To start *an interactive chat* with a model, use `llm chat`:

```
llm chat -m chatgpt
```

```
Chatting with gpt-3.5-turbo
Type 'exit' or 'quit' to exit
Type '!multi' to enter multiple lines, then '!end' to finish
> Tell me a joke about a pelican
Why don't pelicans like to tip waiters?

Because they always have a big bill!
>
```


CONTENTS

2.1 Setup

2.1.1 Installation

Install this tool using `pip`:

```
pip install llm
```

Or using `pipx`:

```
pipx install llm
```

2.1.2 Upgrading to the latest version

If you installed using `pip`:

```
pip install -U llm
```

For `pipx`:

```
pipx upgrade llm
```

For Homebrew:

```
brew upgrade llm
```

If the latest version is not yet available on Homebrew you can upgrade like this instead:

```
llm install -U llm
```

2.1.3 Installing plugins

Plugins can be used to add support for other language models, including models that can run on your own device.

For example, the `llm-gpt4all` plugin adds support for 17 new models that can be installed on your own machine. You can install that like so:

```
llm install llm-gpt4all
```

2.1.4 API key management

Many LLM models require an API key. These API keys can be provided to this tool using several different mechanisms.

You can obtain an API key for OpenAI's language models from [the API keys page](#) on their site.

Saving and using stored keys

The easiest way to store an API key is to use the `llm keys set` command:

```
llm keys set openai
```

You will be prompted to enter the key like this:

```
% llm keys set openai
Enter key:
```

Once stored, this key will be automatically used for subsequent calls to the API:

```
llm "Five ludicrous names for a pet lobster"
```

You can list the names of keys that have been set using this command:

```
llm keys
```

Keys that are stored in this way live in a file called `keys.json`. This file is located at the path shown when you run the following command:

```
llm keys path
```

On macOS this will be `~/Library/Application Support/io.datasette.llm/keys.json`. On Linux it may be something like `~/.config/io.datasette.llm/keys.json`.

Passing keys using the `-key` option

Keys can be passed directly using the `--key` option, like this:

```
llm "Five names for pet weasels" --key sk-my-key-goes-here
```

You can also pass the alias of a key stored in the `keys.json` file. For example, if you want to maintain a personal API key you could add that like this:

```
llm keys set personal
```

And then use it for prompts like so:

```
llm "Five friendly names for a pet skunk" --key personal
```

Keys in environment variables

Keys can also be set using an environment variable. These are different for different models.

For OpenAI models the key will be read from the `OPENAI_API_KEY` environment variable.

The environment variable will be used if no `--key` option is passed to the command and there is not a key configured in `keys.json`

To use an environment variable in place of the `keys.json` key run the prompt like this:

```
llm 'my prompt' --key $OPENAI_API_KEY
```

2.1.5 Configuration

You can configure LLM in a number of different ways.

Setting a custom default model

The model used when calling `llm` without the `-m/--model` option defaults to `gpt-3.5-turbo` - the fastest and least expensive OpenAI model, and the same model family that powers ChatGPT.

You can use the `llm models default` command to set a different default model. For GPT-4 (slower and more expensive, but more capable) run this:

```
llm models default gpt-4
```

You can view the current model by running this:

```
llm models default
```

Any of the supported aliases for a model can be passed to this command.

Setting a custom directory location

This tool stores various files - prompt templates, stored keys, preferences, a database of logs - in a directory on your computer.

On macOS this is `~/Library/Application Support/io.datasette.llm/`.

On Linux it may be something like `~/.config/io.datasette.llm/`.

You can set a custom location for this directory by setting the `LLM_USER_PATH` environment variable:

```
export LLM_USER_PATH=/path/to/my/custom/directory
```

Turning SQLite logging on and off

By default, LLM will log every prompt and response you make to a SQLite database - see [Logging to SQLite](#) for more details.

You can turn this behavior off by default by running:

```
llm logs off
```

Or turn it back on again with:

```
llm logs on
```

Run `llm logs status` to see the current states of the setting.

2.2 Usage

The command to run a prompt is `llm prompt 'your prompt'`. This is the default command, so you can use `llm 'your prompt'` as a shortcut.

2.2.1 Executing a prompt

These examples use the default OpenAI `gpt-3.5-turbo` model, which requires you to first *set an OpenAI API key*.

You can *install LLM plugins* to use models from other providers, including openly licensed models you can run directly on your own computer.

To run a prompt, streaming tokens as they come in:

```
llm 'Ten names for cheesecakes'
```

To disable streaming and only return the response once it has completed:

```
llm 'Ten names for cheesecakes' --no-stream
```

To switch from ChatGPT 3.5 (the default) to GPT-4 Turbo:

```
llm 'Ten names for cheesecakes' -m gpt-4-turbo
```

You can use `-m 4t` as an even shorter shortcut.

Pass `--model <model name>` to use a different model. Run `llm models` to see a list of available models.

You can also send a prompt to standard input, for example:

```
echo 'Ten names for cheesecakes' | llm
```

If you send text to standard input and provide arguments, the resulting prompt will consist of the piped content followed by the arguments:

```
cat myscript.py | llm 'explain this code'
```

Will run a prompt of:

```
<contents of myscript.py> explain this code
```

For models that support them, *system prompts* are a better tool for this kind of prompting.

Some models support options. You can pass these using `-o/--option name value` - for example, to set the temperature to 1.5 run this:

```
llm 'Ten names for cheesecakes' -o temperature 1.5
```

2.2.2 Completion prompts

Some models are completion models - rather than being tuned to respond to chat style prompts, they are designed to complete a sentence or paragraph.

An example of this is the `gpt-3.5-turbo-instruct` OpenAI model.

You can prompt that model the same way as the chat models, but be aware that the prompt format that works best is likely to differ.

```
llm -m gpt-3.5-turbo-instruct 'Reasons to tame a wild beaver:'
```

2.2.3 Continuing a conversation

By default, the tool will start a new conversation each time you run it.

You can opt to continue the previous conversation by passing the `-c/--continue` option:

```
llm 'More names' -c
```

This will re-send the prompts and responses for the previous conversation as part of the call to the language model. Note that this can add up quickly in terms of tokens, especially if you are using expensive models.

`--continue` will automatically use the same model as the conversation that you are continuing, even if you omit the `-m/--model` option.

To continue a conversation that is not the most recent one, use the `--cid/--conversation <id>` option:

```
llm 'More names' --cid 01h53zma5txeby33t1kbe3xk8q
```

You can find these conversation IDs using the `llm logs` command.

2.2.4 Using with a shell

To learn more about your computer's operating system based on the output of `uname -a`, run this:

```
llm "Tell me about my operating system: $(uname -a)"
```

This pattern of using `$(command)` inside a double quoted string is a useful way to quickly assemble prompts.

2.2.5 System prompts

You can use `-s/--system '...'` to set a system prompt.

```
llm 'SQL to calculate total sales by month' \  
  --system 'You are an exaggerated sentient cheesecake that knows SQL and talks about_  
→cheesecake a lot'
```

This is useful for piping content to standard input, for example:

```
curl -s 'https://simonwillison.net/2023/May/15/per-interpreter-gils/' | \  
  llm -s 'Suggest topics for this post as a JSON array'
```

Or to generate a description of changes made to a Git repository since the last commit:

```
git diff | llm -s 'Describe these changes'
```

Different models support system prompts in different ways.

The OpenAI models are particularly good at using system prompts as instructions for how they should process additional input sent as part of the regular prompt.

Other models might use system prompts change the default voice and attitude of the model.

System prompts can be saved as *templates* to create reusable tools. For example, you can create a template called `pytest` like this:

```
llm -s 'write pytest tests for this code' --save pytest
```

And then use the new template like this:

```
cat llm/utils.py | llm -t pytest
```

See *prompt templates* for more.

2.2.6 Starting an interactive chat

The `llm chat` command starts an ongoing interactive chat with a model.

This is particularly useful for models that run on your own machine, since it saves them from having to be loaded into memory each time a new prompt is added to a conversation.

Run `llm chat`, optionally with a `-m model_id`, to start a chat conversation:

```
llm chat -m chatgpt
```

Each chat starts a new conversation. A record of each conversation can be accessed through *the logs*.

For models that support them, you can pass options using `-o/--option`:

```
llm chat -m gpt-4 -o temperature 0.5
```

You can pass a system prompt to be used for your chat conversation:

```
llm chat -m gpt-4 -s 'You are a sentient cheesecake'
```

You can also pass *a template* - useful for creating chat personas that you wish to return to.

Here's how to create a template for your GPT-4 powered cheesecake:

```
llm --system 'You are a sentient cheesecake' -m gpt-4 --save cheesecake
```

Now you can start a new chat with your cheesecake any time you like using this:

```
llm chat -t cheesecake
```

Chatting with gpt-4

Type 'exit' or 'quit' to exit

Type '!multi' to enter multiple lines, then '!end' to finish

> who are you?

I am a sentient cheesecake, meaning I am an artificial intelligence embodied in a dessert form, specifically a cheesecake. However, I don't consume or prepare foods like humans do, I communicate, learn and help answer your queries.

Type quit or exit followed by <enter> to end a chat session.

Sometimes you may want to paste multiple lines of text into a chat at once - for example when debugging an error message.

To do that, type !multi to start a multi-line input. Type or paste your text, then type !end and hit <enter> to finish.

If your pasted text might itself contain a !end line, you can set a custom delimiter using !multi abc followed by !end abc at the end:

Chatting with gpt-4

Type 'exit' or 'quit' to exit

Type '!multi' to enter multiple lines, then '!end' to finish

> !multi custom-end

Explain this error:

```
File "/opt/homebrew/Caskroom/miniconda/base/lib/python3.10/urllib/request.py", line
↳1391, in https_open
    return self.do_open(http.client.HTTPSConnection, req,
File "/opt/homebrew/Caskroom/miniconda/base/lib/python3.10/urllib/request.py", line
↳1351, in do_open
    raise URLError(err)
urllib.error.URLError: <urlopen error [Errno 8] nodename nor servname provided, or not
↳known>
```

!end custom-end

2.2.7 Listing available models

The `llm models` command lists every model that can be used with LLM, along with their aliases. This includes models that have been installed using *plugins*.

```
llm models
```

Example output:

```
OpenAI Chat: gpt-3.5-turbo (aliases: 3.5, chatgpt)
OpenAI Chat: gpt-3.5-turbo-16k (aliases: chatgpt-16k, 3.5-16k)
OpenAI Chat: gpt-4 (aliases: 4, gpt4)
OpenAI Chat: gpt-4-32k (aliases: 4-32k)
PaLM 2: chat-bison-001 (aliases: palm, palm2)
```

Add `--options` to also see documentation for the options supported by each model:

```
llm models --options
```

Output:

```
OpenAI Chat: gpt-3.5-turbo (aliases: 3.5, chatgpt)
  temperature: float
    What sampling temperature to use, between 0 and 2. Higher values like
    0.8 will make the output more random, while lower values like 0.2 will
    make it more focused and deterministic.
  max_tokens: int
    Maximum number of tokens to generate.
  top_p: float
    An alternative to sampling with temperature, called nucleus sampling,
    where the model considers the results of the tokens with top_p
    probability mass. So 0.1 means only the tokens comprising the top 10%
    probability mass are considered. Recommended to use top_p or
    temperature but not both.
  frequency_penalty: float
    Number between -2.0 and 2.0. Positive values penalize new tokens based
    on their existing frequency in the text so far, decreasing the model's
    likelihood to repeat the same line verbatim.
  presence_penalty: float
    Number between -2.0 and 2.0. Positive values penalize new tokens based
    on whether they appear in the text so far, increasing the model's
    likelihood to talk about new topics.
  stop: str
    A string where the API will stop generating further tokens.
  logit_bias: dict, str
    Modify the likelihood of specified tokens appearing in the completion.
    Pass a JSON string like '{"1712":-100, "892":-100, "1489":-100}'
  seed: int
    Integer seed to attempt to sample deterministically
OpenAI Chat: gpt-3.5-turbo-16k (aliases: chatgpt-16k, 3.5-16k)
  temperature: float
  max_tokens: int
  top_p: float
  frequency_penalty: float
```

(continues on next page)

(continued from previous page)

```

presence_penalty: float
stop: str
logit_bias: dict, str
seed: int
OpenAI Chat: gpt-4 (aliases: 4, gpt4)
  temperature: float
  max_tokens: int
  top_p: float
  frequency_penalty: float
  presence_penalty: float
  stop: str
  logit_bias: dict, str
  seed: int
OpenAI Chat: gpt-4-1106-preview (aliases: gpt-4-turbo, 4-turbo, 4t)
  temperature: float
  max_tokens: int
  top_p: float
  frequency_penalty: float
  presence_penalty: float
  stop: str
  logit_bias: dict, str
  seed: int
OpenAI Chat: gpt-4-32k (aliases: 4-32k)
  temperature: float
  max_tokens: int
  top_p: float
  frequency_penalty: float
  presence_penalty: float
  stop: str
  logit_bias: dict, str
  seed: int
OpenAI Completion: gpt-3.5-turbo-instruct (aliases: 3.5-instruct, chatgpt-instruct)
  temperature: float
    What sampling temperature to use, between 0 and 2. Higher values like
    0.8 will make the output more random, while lower values like 0.2 will
    make it more focused and deterministic.
  max_tokens: int
    Maximum number of tokens to generate.
  top_p: float
    An alternative to sampling with temperature, called nucleus sampling,
    where the model considers the results of the tokens with top_p
    probability mass. So 0.1 means only the tokens comprising the top 10%
    probability mass are considered. Recommended to use top_p or
    temperature but not both.
  frequency_penalty: float
    Number between -2.0 and 2.0. Positive values penalize new tokens based
    on their existing frequency in the text so far, decreasing the model's
    likelihood to repeat the same line verbatim.
  presence_penalty: float
    Number between -2.0 and 2.0. Positive values penalize new tokens based
    on whether they appear in the text so far, increasing the model's
    likelihood to talk about new topics.

```

(continues on next page)

(continued from previous page)

```

stop: str
  A string where the API will stop generating further tokens.
logit_bias: dict, str
  Modify the likelihood of specified tokens appearing in the completion.
  Pass a JSON string like '{"1712":-100, "892":-100, "1489":-100}'
seed: int
  Integer seed to attempt to sample deterministically
logprobs: int
  Include the log probabilities of most likely N per token

```

When running a prompt you can pass the full model name or any of the aliases to the `-m/--model` option:

```

llm -m chatgpt-16k \
  'As many names for cheesecakes as you can think of, with detailed descriptions'

```

2.3 Other models

LLM supports OpenAI models by default. You can install *plugins* to add support for other models. You can also add additional OpenAI-API-compatible models *using a configuration file*.

2.3.1 Installing and using a local model

LLM plugins can provide local models that run on your machine.

To install **llm-gpt4all**, providing 17 models from the [GPT4All](#) project, run this:

```
llm install llm-gpt4all
```

Run `llm models` to see the expanded list of available models.

To run a prompt through one of the models from GPT4All specify it using `-m/--model`:

```
llm -m orca-mini-3b-gguf2-q4_0 'What is the capital of France?'
```

The model will be downloaded and cached the first time you use it.

Check the *plugin directory* for the latest list of available plugins for other models.

2.3.2 Adding more OpenAI models

OpenAI occasionally release new models with new names. LLM aims to ship new releases to support these, but you can also configure them directly, by adding them to a `extra-openai-models.yaml` configuration file.

Run this command to find the directory in which this file should be created:

```
dirname "$(llm logs path)"
```

On my Mac laptop I get this:

```
~/Library/Application Support/io.datasette.llm
```

Create a file in that directory called `extra-openai-models.yaml`.

Let's say OpenAI have just released the `gpt-3.5-turbo-0613` model and you want to use it, despite LLM not yet shipping support. You could configure that by adding this to the file:

```
- model_id: gpt-3.5-turbo-0613
  aliases: ["0613"]
```

The `model_id` is the identifier that will be recorded in the LLM logs. You can use this to specify the model, or you can optionally include a list of aliases for that model.

If the model is a completion model (such as `gpt-3.5-turbo-instruct`) add `completion: true` to the configuration.

With this configuration in place, the following command should run a prompt against the new model:

```
llm -m 0613 'What is the capital of France?'
```

Run `llm models` to confirm that the new model is now available:

```
llm models
```

Example output:

```
OpenAI Chat: gpt-3.5-turbo (aliases: 3.5, chatgpt)
OpenAI Chat: gpt-3.5-turbo-16k (aliases: chatgpt-16k, 3.5-16k)
OpenAI Chat: gpt-4 (aliases: 4, gpt4)
OpenAI Chat: gpt-4-32k (aliases: 4-32k)
OpenAI Chat: gpt-3.5-turbo-0613 (aliases: 0613)
```

Running `llm logs -n 1` should confirm that the prompt and response has been correctly logged to the database.

2.3.3 OpenAI-compatible models

Projects such as [LocalAI](#) offer a REST API that imitates the OpenAI API but can be used to run other models, including models that can be installed on your own machine. These can be added using the same configuration mechanism.

The `model_id` is the name LLM will use for the model. The `model_name` is the name which needs to be passed to the API - this might differ from the `model_id`, especially if the `model_id` could potentially clash with other installed models.

The `api_base` key can be used to point the OpenAI client library at a different API endpoint.

To add the `orca-mini-3b` model hosted by a local installation of [LocalAI](#), add this to your `extra-openai-models.yaml` file:

```
- model_id: orca-openai-compat
  model_name: orca-mini-3b.ggmlv3
  api_base: "http://localhost:8080"
```

If the `api_base` is set, the existing configured `openai` API key will not be sent by default.

You can set `api_key_name` to the name of a key stored using the [API key management](#) feature.

Add `completion: true` if the model is a completion model that uses a `/completion` as opposed to a `/completion/chat` endpoint.

Having configured the model like this, run `llm models` to check that it installed correctly. You can then run prompts against it like so:

```
llm -m orca-openai-compat 'What is the capital of France?'
```

And confirm they were logged correctly with:

```
llm logs -n 1
```

Extra HTTP headers

Some providers such as openrouter.ai may require the setting of additional HTTP headers. You can set those using the `headers`: key like this:

```
- model_id: claude
  model_name: anthropic/claude-2
  api_base: "https://openrouter.ai/api/v1"
  api_key_name: openrouter
  headers:
    HTTP-Referer: "https://llm.datasette.io/"
    X-Title: LLM
```

2.4 Embeddings

Embedding models allow you to take a piece of text - a word, sentence, paragraph or even a whole article, and convert that into an array of floating point numbers.

This floating point array is called an “embedding vector”, and works as a numerical representation of the semantic meaning of the content in a many-multi-dimensional space.

By calculating the distance between embedding vectors, we can identify which content is semantically “nearest” to other content.

This can be used to build features like related article lookups. It can also be used to build semantic search, where a user can search for a phrase and get back results that are semantically similar to that phrase even if they do not share any exact keywords.

Some embedding models like [CLIP](#) can even work against binary files such as images. These can be used to search for images that are similar to other images, or to search for images that are semantically similar to a piece of text.

LLM supports multiple embedding models through [plugins](#). Once installed, an embedding model can be used on the command-line or via the Python API to calculate and store embeddings for content, and then to perform similarity searches against those embeddings.

See [LLM now provides tools for working with embeddings](#) for an extended explanation of embeddings, why they are useful and what you can do with them.

2.4.1 Embedding with the CLI

LLM provides command-line utilities for calculating and storing embeddings for pieces of content.

llm embed

The `llm embed` command can be used to calculate embedding vectors for a string of content. These can be returned directly to the terminal, stored in a SQLite database, or both.

Returning embeddings to the terminal

The simplest way to use this command is to pass content to it using the `-c/--content` option, like this:

```
llm embed -c 'This is some content' -m ada-002
```

`-m ada-002` specifies the OpenAI `ada-002` model. You will need to have set an OpenAI API key using `llm keys set openai` for this to work.

You can install plugins to access other models. The `llm-sentence-transformers` plugin can be used to run models on your own laptop, such as the `MiniLM-L6` model:

```
llm install llm-sentence-transformers
llm embed -c 'This is some content' -m sentence-transformers/all-MiniLM-L6-v2
```

The `llm embed` command returns a JSON array of floating point numbers directly to the terminal:

```
[0.123, 0.456, 0.789...]
```

You can omit the `-m/--model` option if you set a *default embedding model*.

LLM also offers a binary storage format for embeddings, described in *embeddings storage format*.

You can output embeddings using that format as raw bytes using `--format blob`, or in hexadecimal using `--format hex`, or in Base64 using `--format base64`:

```
llm embed -c 'This is some content' -m ada-002 --format base64
```

This outputs:

```
8NGzPFtdgTqHcZw7aUT6u+++WrwwpZo8XbSxv...
```

Some models such as `llm-clip` can run against binary data. You can pass in binary data using the `-i` and `--binary` options:

```
llm embed --binary -m clip -i image.jpg
```

Or from standard input like this:

```
cat image.jpg | llm embed --binary -m clip -i -
```

Storing embeddings in SQLite

Embeddings are much more useful if you store them somewhere, so you can calculate similarity scores between different embeddings later on.

LLM includes the concept of a **collection** of embeddings. A collection groups together a set of stored embeddings created using the same model, each with a unique ID within that collection.

Embeddings also store a hash of the content that was embedded. This hash is later used to avoid calculating duplicate embeddings for the same content.

First, we'll set a default model so we don't have to keep repeating it:

```
llm embed-models default ada-002
```

The `llm embed` command can store results directly in a named collection like this:

```
llm embed quotations philkarlton-1 -c \  
'There are only two hard things in Computer Science: cache invalidation and naming_\  
↳things'
```

This stores the given text in the `quotations` collection under the key `philkarlton-1`.

You can also pipe content to standard input, like this:

```
cat one.txt | llm embed files one
```

This will store the embedding for the contents of `one.txt` in the `files` collection under the key `one`.

A collection will be created the first time you mention it.

Collections have a fixed embedding model, which is the model that was used for the first embedding stored in that collection.

In the above example this would have been the default embedding model at the time that the command was run.

The following example stores the embedding for the string “my happy hound” in a collection called `phrases` under the key `hound` and using the model `ada-002`:

```
llm embed phrases hound -m ada-002 -c 'my happy hound'
```

By default, the SQLite database used to store embeddings is the `embeddings.db` in the user content directory managed by LLM.

You can see the path to this directory by running `llm collections path`.

You can store embeddings in a different SQLite database by passing a path to it using the `-d/--database` option to `llm embed`. If this file does not exist yet the command will create it:

```
llm embed phrases hound -d my-embeddings.db -c 'my happy hound'
```

This creates a database file called `my-embeddings.db` in the current directory.

Storing content and metadata

By default, only the entry ID and the embedding vector are stored in the database table.

You can store a copy of the original text in the `content` column by passing the `--store` option:

```
llm embed phrases hound -c 'my happy hound' --store
```

You can also store a JSON object containing arbitrary metadata in the `metadata` column by passing the `--metadata` option. This example uses both `--store` and `--metadata` options:

```
llm embed phrases hound \
-m ada-002 \
-c 'my happy hound' \
--metadata '{"name": "Hound"}' \
--store
```

Data stored in this way will be returned by calls to `llm similar`, for example:

```
llm similar phrases -c 'hound'
```

```
{"id": "hound", "score": 0.8484683588631485, "content": "my happy hound", "metadata": {
  ↪ "name": "Hound"}}
```

llm embed-multi

The `llm embed` command embeds a single string at a time.

`llm embed-multi` can be used to embed multiple strings at once, taking advantage of any efficiencies that the embedding model may provide when processing multiple strings.

This command can be called in one of three ways:

1. With a CSV, TSV, JSON or newline-delimited JSON file
2. With a SQLite database and a SQL query
3. With one or more paths to directories, each accompanied by a glob pattern

All three mechanisms support these options:

- `-m model_id` to specify the embedding model to use
- `-d database.db` to specify a different database file to store the embeddings in
- `--store` to store the original content in the embeddings table in addition to the embedding vector
- `--prefix` to prepend a prefix to the stored ID of each item
- `--batch-size SIZE` to process embeddings in batches of the specified size

Embedding data from a CSV, TSV or JSON file

You can embed data from a CSV, TSV or JSON file by passing that file to the command as the second option, after the collection name.

Your file must contain at least two columns. The first one is expected to contain the ID of the item, and any subsequent columns will be treated as containing content to be embedded.

An example CSV file might look like this:

```
id,content
one,This is the first item
two,This is the second item
```

TSV would use tabs instead of commas.

JSON files can be structured like this:

```
[
  {"id": "one", "content": "This is the first item"},
  {"id": "two", "content": "This is the second item"}
]
```

Or as newline-delimited JSON like this:

```
{"id": "one", "content": "This is the first item"}
{"id": "two", "content": "This is the second item"}
```

In each of these cases the file can be passed to `llm embed-multi` like this:

```
llm embed-multi items mydata.csv
```

The first argument is the name of the collection, the second is the filename.

You can also pipe content to standard input of the tool using `-`:

```
cat mydata.json | llm embed-multi items -
```

LLM will attempt to detect the format of your data automatically. If this doesn't work you can specify the format using the `--format` option. This is required if you are piping newline-delimited JSON to standard input.

```
cat mydata.json | llm embed-multi items - --format nl
```

Other supported `--format` options are `csv`, `tsv` and `json`.

This example embeds the data from a JSON file in a collection called `items` in database called `docs.db` using the `ada-002` model and stores the original content in the `embeddings` table as well, adding a prefix of `my-items/` to each ID:

```
llm embed-multi items mydata.json \
-d docs.db \
-m ada-002 \
--prefix my-items/ \
--store
```

Embedding data from a SQLite database

You can embed data from a SQLite database using `--sql`, optionally combined with `--attach` to attach an additional database.

If you are storing embeddings in the same database as the source data, you can do this:

```
llm embed-multi docs \
-d docs.db \
--sql 'select id, title, content from documents' \
-m ada-002
```

The `docs.db` database here contains a `documents` table, and we want to embed the `title` and `content` columns from that table and store the results back in the same database.

To load content from a database other than the one you are using to store embeddings, attach it with the `--attach` option and use `alias.table` in your SQLite query:

```
llm embed-multi docs \
-d embeddings.db \
--attach other other.db \
--sql 'select id, title, content from other.documents' \
-m ada-002
```

Embedding data from files in directories

LLM can embed the content of every text file in a specified directory, using the file's path and name as the ID.

Consider a directory structure like this:

```
docs/aliases.md
docs/contributing.md
docs/embeddings/binary.md
docs/embeddings/cli.md
docs/embeddings/index.md
docs/index.md
docs/logging.md
docs/plugins/directory.md
docs/plugins/index.md
```

To embed all of those documents, you can run the following:

```
llm embed-multi documentation \
-m ada-002 \
--files docs '**/*.md' \
-d documentation.db \
--store
```

Here `--files docs '**/*.md'` specifies that the `docs` directory should be scanned for files matching the `**/*.md` glob pattern - which will match Markdown files in any nested directory.

The result of the above command is a `embeddings` table with the following IDs:

```
aliases.md
contributing.md
embeddings/binary.md
embeddings/cli.md
embeddings/index.md
index.md
logging.md
plugins/directory.md
plugins/index.md
```

Each corresponding to embedded content for the file in question.

The `--prefix` option can be used to add a prefix to each ID:

```
llm embed-multi documentation \
-m ada-002 \
--files docs '**/*.md' \
-d documentation.db \
--store \
--prefix llm-docs/
```

This will result in the following IDs instead:

```
llm-docs/aliases.md
llm-docs/contributing.md
llm-docs/embeddings/binary.md
llm-docs/embeddings/cli.md
llm-docs/embeddings/index.md
llm-docs/index.md
llm-docs/logging.md
llm-docs/plugins/directory.md
llm-docs/plugins/index.md
```

Files are assumed to be `utf-8`, but LLM will fall back to `latin-1` if it encounters an encoding error. You can specify a different set of encodings using the `--encoding` option.

This example will try `utf-16` first and then `mac_roman` before falling back to `latin-1`:

```
llm embed-multi documentation \
-m ada-002 \
--files docs '**/*.md' \
-d documentation.db \
--encoding utf-16 \
--encoding mac_roman \
--encoding latin-1
```

If a file cannot be read it will be logged to standard error but the script will keep on running.

If you are embedding binary content such as images for use with CLIP, add the `--binary` option:

```
llm embed-multi photos \
-m clip \
--files photos/ '*.jpeg' --binary
```

llm similar

The `llm similar` command searches a collection of embeddings for the items that are most similar to a given or item ID.

This currently uses a slow brute-force approach which does not scale well to large collections. See [issue 216](#) for plans to add a more scalable approach via vector indexes provided by plugins.

To search the `quotations` collection for items that are semantically similar to `'computer science'`:

```
llm similar quotations -c 'computer science'
```

This embeds the provided string and returns a newline-delimited list of JSON objects like this:

```
{"id": "philkarlton-1", "score": 0.8323904531677017, "content": null, "metadata": null}
```

You can compare against text stored in a file using `-i filename`:

```
llm similar quotations -i one.txt
```

Or feed text to standard input using `-i -`:

```
echo 'computer science' | llm similar quotations -i -
```

When using a model like CLIP, you can find images similar to an input image using `-i filename` with `--binary`:

```
llm similar photos -i image.jpg --binary
```

llm embed-models

To list all available embedding models, including those provided by plugins, run this command:

```
llm embed-models
```

The output should look something like this:

```
ada-002 (aliases: ada)
sentence-transformers/all-MiniLM-L6-v2 (aliases: all-MiniLM-L6-v2)
```

llm embed-models default

This command can be used to get and set the default embedding model.

This will return the name of the current default model:

```
llm embed-models default
```

You can set a different default like this:

```
llm embed-models default ada-002
```

This will set the default model to OpenAI's `ada-002` model.

Any of the supported aliases for a model can be passed to this command.

You can unset the default model using `--remove-default`:

```
llm embed-models default --remove-default
```

When no default model is set, the `llm embed` and `llm embed-multi` commands will require that a model is specified using `-m/--model`.

llm collections list

To list all of the collections in the embeddings database, run this command:

```
llm collections list
```

Add `--json` for JSON output:

```
llm collections list --json
```

Add `-d/--database` to specify a different database file:

```
llm collections list -d my-embeddings.db
```

llm collections delete

To delete a collection from the database, run this:

```
llm collections delete collection-name
```

Pass `-d` to specify a different database file:

```
llm collections delete collection-name -d my-embeddings.db
```

2.4.2 Using embeddings from Python

You can load an embedding model using its model ID or alias like this:

```
import llm

embedding_model = llm.get_embedding_model("ada-002")
```

To embed a string, returning a Python list of floating point numbers, use the `.embed()` method:

```
vector = embedding_model.embed("my happy hound")
```

If the embedding model can handle binary input, you can call `.embed()` with a byte string instead. You can check the `supports_binary` property to see if this is supported:

```
if embedding_model.supports_binary:
    vector = embedding_model.embed(open("my-image.jpg", "rb").read())
```

The `embedding_model.supports_text` property indicates if the model supports text input.

Many embeddings models are more efficient when you embed multiple strings or binary strings at once. To embed multiple strings at once, use the `.embed_multi()` method:

```
vectors = list(embedding_model.embed_multi(["my happy hound", "my dissatisfied cat"]))
```

This returns a generator that yields one embedding vector per string.

Embeddings are calculated in batches. By default all items will be processed in a single batch, unless the underlying embedding model has defined its own preferred batch size. You can pass a custom batch size using `batch_size=N`, for example:

```
vectors = list(embedding_model.embed_multi(lines_from_file, batch_size=20))
```

Working with collections

The `llm.Collection` class can be used to work with **collections** of embeddings from Python code.

A collection is a named group of embedding vectors, each stored along with their IDs in a SQLite database table.

To work with embeddings in this way you will need an instance of a `sqlite-utils Database` object. You can then pass that to the `llm.Collection` constructor along with the unique string name of the collection and the ID of the embedding model you will be using with that collection:

```
import sqlite_utils
import llm

# This collection will use an in-memory database that will be
# discarded when the Python process exits
collection = llm.Collection("entries", model_id="ada-002")

# Or you can persist the database to disk like this:
db = sqlite_utils.Database("my-embeddings.db")
collection = llm.Collection("entries", db, model_id="ada-002")

# You can pass a model directly using model= instead of model_id=
embedding_model = llm.get_embedding_model("ada-002")
collection = llm.Collection("entries", db, model=embedding_model)
```

If the collection already exists in the database you can omit the `model` or `model_id` argument - the model ID will be read from the collections table.

To embed a single string and store it in the collection, use the `embed()` method:

```
collection.embed("hound", "my happy hound")
```

This stores the embedding for the string “my happy hound” in the `entries` collection under the key `hound`.

Add `store=True` to store the text content itself in the database table along with the embedding vector.

To attach additional metadata to an item, pass a JSON-compatible dictionary as the `metadata=` argument:

```
collection.embed("hound", "my happy hound", metadata={"name": "Hound"}, store=True)
```

This additional metadata will be stored as JSON in the `metadata` column of the embeddings database table.

Storing embeddings in bulk

The `collection.embed_multi()` method can be used to store embeddings for multiple items at once. This can be more efficient for some embedding models.

```
collection.embed_multi(  
    [  
        ("hound", "my happy hound"),  
        ("cat", "my dissatisfied cat"),  
    ],  
    # Add this to store the strings in the content column:  
    store=True,  
)
```

To include metadata to be stored with each item, call `embed_multi_with_metadata()`:

```
collection.embed_multi_with_metadata(  
    [  
        ("hound", "my happy hound", {"name": "Hound"}),  
        ("cat", "my dissatisfied cat", {"name": "Cat"}),  
    ],  
    # This can also take the store=True argument:  
    store=True,  
)
```

The `batch_size=` argument defaults to 100, and will be used unless the embedding model itself defines a lower batch size. You can adjust this if you are having trouble with memory while embedding large collections:

```
collection.embed_multi(  
    (  
        (i, line)  
        for i, line in enumerate(lines_in_file)  
    ),  
    batch_size=10  
)
```

Collection class reference

A collection instance has the following properties and methods:

- `id` - the integer ID of the collection in the database
- `name` - the string name of the collection (unique in the database)
- `model_id` - the string ID of the embedding model used for this collection
- `model()` - returns the `EmbeddingModel` instance, based on that `model_id`
- `count()` - returns the integer number of items in the collection
- `embed(id: str, text: str, metadata: dict=None, store: bool=False)` - embeds the given string and stores it in the collection under the given ID. Can optionally include metadata (stored as JSON) and store the text content itself in the database table.
- `embed_multi(entries: Iterable, store: bool=False, batch_size: int=100)` - see above

- `embed_multi_with_metadata(entries: Iterable, store: bool=False, batch_size: int=100)` - see above
- `similar(query: str, number: int=10)` - returns a list of entries that are most similar to the embedding of the given query string
- `similar_by_id(id: str, number: int=10)` - returns a list of entries that are most similar to the embedding of the item with the given ID
- `similar_by_vector(vector: List[float], number: int=10, skip_id: str=None)` - returns a list of entries that are most similar to the given embedding vector, optionally skipping the entry with the given ID
- `delete()` - deletes the collection and its embeddings from the database

There is also a `Collection.exists(db, name)` class method which returns a boolean value and can be used to determine if a collection exists or not in a database:

```
if Collection.exists(db, "entries"):
    print("The entries collection exists")
```

Retrieving similar items

Once you have populated a collection of embeddings you can retrieve the entries that are most similar to a given string using the `similar()` method.

This method uses a brute force approach, calculating distance scores against every document. This is fine for small collections, but will not scale to large collections. See [issue 216](#) for plans to add a more scalable approach via vector indexes provided by plugins.

```
for entry in collection.similar("hound"):
    print(entry.id, entry.score)
```

The string will first be embedded using the model for the collection.

The entry object returned is an object with the following properties:

- `id` - the string ID of the item
- `score` - the floating point similarity score between the item and the query string
- `content` - the string text content of the item, if it was stored - or `None`
- `metadata` - the dictionary (from JSON) metadata for the item, if it was stored - or `None`

This defaults to returning the 10 most similar items. You can change this by passing a different `number=` argument:

```
for entry in collection.similar("hound", number=5):
    print(entry.id, entry.score)
```

The `similar_by_id()` method takes the ID of another item in the collection and returns the most similar items to that one, based on the embedding that has already been stored for it:

```
for entry in collection.similar_by_id("cat"):
    print(entry.id, entry.score)
```

The item itself is excluded from the results.

SQL schema

Here's the SQL schema used by the embeddings database:

```
CREATE TABLE [collections] (
  [id] INTEGER PRIMARY KEY,
  [name] TEXT,
  [model] TEXT
)
CREATE TABLE "embeddings" (
  [collection_id] INTEGER REFERENCES [collections]([id]),
  [id] TEXT,
  [embedding] BLOB,
  [content] TEXT,
  [content_blob] BLOB,
  [content_hash] BLOB,
  [metadata] TEXT,
  [updated] INTEGER,
  PRIMARY KEY ([collection_id], [id])
)
```

2.4.3 Writing plugins to add new embedding models

Read the *plugin tutorial* for details on how to develop and package a plugin.

This page shows an example plugin that implements and registers a new embedding model.

There are two components to an embedding model plugin:

1. An implementation of the `register_embedding_models()` hook, which takes a `register` callback function and calls it to register the new model with the LLM plugin system.
2. A class that extends the `llm.EmbeddingModel` abstract base class.

The only required method on this class is `embed_batch(texts)`, which takes an iterable of strings and returns an iterator over lists of floating point numbers.

The following example uses the `sentence-transformers` package to provide access to the `MiniLM-L6` embedding model.

```
import llm
from sentence_transformers import SentenceTransformer

@llm.hookimpl
def register_embedding_models(register):
    model_id = "sentence-transformers/all-MiniLM-L6-v2"
    register(SentenceTransformerModel(model_id, model_id), aliases=("all-MiniLM-L6-v2",))

class SentenceTransformerModel(llm.EmbeddingModel):
    def __init__(self, model_id, model_name):
        self.model_id = model_id
        self.model_name = model_name
        self._model = None
```

(continues on next page)

(continued from previous page)

```
def embed_batch(self, texts):
    if self._model is None:
        self._model = SentenceTransformer(self.model_name)
    results = self._model.encode(texts)
    return (list(map(float, result)) for result in results)
```

Once installed, the model provided by this plugin can be used with the `llm embed` command like this:

```
cat file.txt | llm embed -m sentence-transformers/all-MiniLM-L6-v2
```

Or via its registered alias like this:

```
cat file.txt | llm embed -m all-MiniLM-L6-v2
```

`llm-sentence-transformers` is a complete example of a plugin that provides an embedding model.

Execute Jina embeddings with a CLI using `llm-embed-jina` talks through a similar process to add support for the Jina embeddings models.

Embedding binary content

If your model can embed binary content, use the `supports_binary` property to indicate that:

```
class ClipEmbeddingModel(llm.EmbeddingModel):
    model_id = "clip"
    supports_binary = True
    supports_text = True
```

`supports_text` defaults to `True` and so is not necessary here. You can set it to `False` if your model only supports binary data.

If your model accepts binary, your `.embed_batch()` model may be called with a list of Python bytestrings. These may be mixed with regular strings if the model accepts both types of input.

`llm-clip` is an example of a model that can embed both binary and text content.

2.4.4 Embedding storage format

The default output format of the `llm embed` command is a JSON array of floating point numbers.

LLM stores embeddings in space-efficient format: a little-endian binary sequences of 32-bit floating point numbers, each represented using 4 bytes.

These are stored in a BLOB column in a SQLite database.

The following Python functions can be used to convert between this format and an array of floating point numbers:

```
import struct

def encode(values):
    return struct.pack("<" + "f" * len(values), *values)

def decode(binary):
    return struct.unpack("<" + "f" * (len(binary) // 4), binary)
```

These functions are available as `llm.encode()` and `llm.decode()`.

If you are using `NumPy` you can decode one of these binary values like this:

```
import numpy as np
numpy_array = np.frombuffer(value, "<f4")
```

The `<f4` format string here ensures `NumPy` will treat the data as a little-endian sequence of 32-bit floats.

2.5 Plugins

LLM plugins can enhance LLM by making alternative Large Language Models available, either via API or by running the models locally on your machine.

Plugins can also add new commands to the `llm` CLI tool.

The *plugin directory* lists available plugins that you can install and use.

Writing a plugin to support a new model describes how to build a new plugin in detail.

2.5.1 Installing plugins

Plugins must be installed in the same virtual environment as LLM itself.

You can find names of plugins to install in the *plugin directory*

Use the `llm install` command (a thin wrapper around `pip install`) to install plugins in the correct environment:

```
llm install llm-gpt4all
```

Plugins can be uninstalled with `llm uninstall`:

```
llm uninstall llm-gpt4all -y
```

The `-y` flag skips asking for confirmation.

You can see additional models that have been added by plugins by running:

```
llm models
```

Or add `--options` to include details of the options available for each model:

```
llm models --options
```

To run a prompt against a newly installed model, pass its name as the `-m/--model` option:

```
llm -m orca-mini-3b-gguf2-q4_0 'What is the capital of France?'
```

Listing installed plugins

Run `llm plugins` to list installed plugins:

```
llm plugins
```

```
[
  {
    "name": "llm-mpt30b",
    "hooks": [
      "register_commands",
      "register_models"
    ],
    "version": "0.1"
  },
  {
    "name": "llm-palm",
    "hooks": [
      "register_commands",
      "register_models"
    ],
    "version": "0.1"
  },
  {
    "name": "llm.default_plugins.openai_models",
    "hooks": [
      "register_commands",
      "register_models"
    ]
  },
  {
    "name": "llm-gpt4all",
    "hooks": [
      "register_models"
    ],
    "version": "0.1"
  }
]
```

Running with a subset of plugins

By default, LLM will load all plugins that are installed in the same virtual environment as LLM itself.

You can control the set of plugins that is loaded using the `LLM_LOAD_PLUGINS` environment variable.

Set that to the empty string to disable all plugins:

```
LLM_LOAD_PLUGINS='' llm ...
```

Or to a comma-separated list of plugin names to load only those plugins:

```
LLM_LOAD_PLUGINS='llm-gpt4all,llm-cluster' llm ...
```

You can use the `llm plugins` command to check that it is working correctly:

```
LLM_LOAD_PLUGINS=' ' llm plugins
```

2.5.2 Plugin directory

The following plugins are available for LLM. Here's *how to install them*.

Local models

These plugins all help you run LLMs directly on your own computer:

- **llm-mlc** can run local models released by the [MLC project](#), including models that can take advantage of the GPU on Apple Silicon M1/M2 devices.
- **llm-llama-cpp** uses `llama.cpp` to run models published in the GGML format.
- **llm-gpt4all** adds support for various models released by the [GPT4All](#) project that are optimized to run locally on your own machine. These models include versions of Vicuna, Orca, Falcon and MPT - here's [a full list of models](#).
- **llm-mpt30b** adds support for the [MPT-30B](#) local model.

Remote APIs

These plugins can be used to interact with remotely hosted models via their API:

- **llm-palm** adds support for Google's [PaLM 2 model](#).
- **llm-replicate** adds support for remote models hosted on [Replicate](#), including Llama 2 from Meta AI.
- **llm-claude** by Tom Viner adds support for Claude and Claude Instant by Anthropic.
- **llm-openrouter** provides access to models hosted on [OpenRouter](#).
- **llm-anyscale-endpoints** supports models hosted on the [Anyscale Endpoints](#) platform, including Llama 2 70B.
- **llm-cohere** by Alistair Shepherd provides `cohere-generate` and `cohere-summarize` API models, powered by [Cohere](#).
- **llm-bedrock-anthropic** adds support for Claude and Claude Instant by Anthropic via Amazon Bedrock.

If an API model host provides an OpenAI-compatible API you can also [configure LLM to talk to it](#) without needing an extra plugin.

Embedding models

Embedding models are models that can be used to generate and store embedding vectors for text.

- **llm-sentence-transformers** adds support for embeddings using the [sentence-transformers](#) library, which provides access to [a wide range](#) of embedding models.
- **llm-clip** provides the [CLIP](#) model, which can be used to embed images and text in the same vector space, enabling text search against images. See [Build an image search engine with llm-clip](#) for more on this plugin.
- **llm-embed-jina** provides Jina AI's [8K text embedding models](#).

Extra commands

- **llm-python** adds a `llm python` command for running a Python interpreter in the same virtual environment as LLM. This is useful for debugging, and also provides a convenient way to interact with the LLM *Python API* if you installed LLM using Homebrew or `pipx`.
- **llm-cluster** adds a `llm cluster` command for calculating clusters for a collection of embeddings. Calculated clusters can then be passed to a Large Language Model to generate a summary description.

Just for fun

- **llm-markov** adds a simple model that generates output using a [Markov chain](#). This example is used in the tutorial [Writing a plugin to support a new model](#).

2.5.3 Plugin hooks

Plugins use **plugin hooks** to customize LLM's behavior. These hooks are powered by the [Pluggy plugin system](#).

Each plugin can implement one or more hooks using the `@hookimpl` decorator against one of the hook function names described on this page.

LLM imitates the Datasette plugin system. The [Datasette plugin documentation](#) describes how plugins work.

`register_commands(cli)`

This hook adds new commands to the `llm` CLI tool - for example `llm extra-command`.

This example plugin adds a new `hello-world` command that prints "Hello world!":

```
from llm import hookimpl
import click

@hookimpl
def register_commands(cli):
    @cli.command(name="hello-world")
    def hello_world():
        "Print hello world"
        click.echo("Hello world!")
```

This new command will be added to `llm --help` and can be run using `llm hello-world`.

`register_models(register)`

This hook can be used to register one or more additional models.

```
import llm

@llm.hookimpl
def register_models(register):
    register>HelloWorld())

class>HelloWorld(llm.Model):
    model_id = "helloworld"
```

(continues on next page)

(continued from previous page)

```
def execute(self, prompt, stream, response):  
    return ["hello world"]
```

Writing a plugin to support a new model describes how to use this hook in detail.

2.5.4 Writing a plugin to support a new model

This tutorial will walk you through developing a new plugin for LLM that adds support for a new Large Language Model.

We will be developing a plugin that implements a simple [Markov chain](#) to generate words based on an input string. Markov chains are not technically large language models, but they provide a useful exercise for demonstrating how the LLM tool can be extended through plugins.

The initial structure of the plugin

First create a new directory with the name of your plugin - it should be called something like `llm-markov`.

```
mkdir llm-markov  
cd llm-markov
```

In that directory create a file called `llm_markov.py` containing this:

```
import llm  
  
@llm.hookimpl  
def register_models(register):  
    register(Markov())  
  
class Markov(llm.Model):  
    model_id = "markov"  
  
    def execute(self, prompt, stream, response, conversation):  
        return ["hello world"]
```

The `def register_models()` function here is called by the plugin system (thanks to the `@hookimpl` decorator). It uses the `register()` function passed to it to register an instance of the new model.

The `Markov` class implements the model. It sets a `model_id` - an identifier that can be passed to `llm -m` in order to identify the model to be executed.

The logic for executing the model goes in the `execute()` method. We'll extend this to do something more useful in a later step.

Next, create a `pyproject.toml` file. This is necessary to tell LLM how to load your plugin:

```
[project]  
name = "llm-markov"  
version = "0.1"  
  
[project.entry-points.llm]  
markov = "llm_markov"
```

This is the simplest possible configuration. It defines a plugin name and provides an [entry point](#) for llm telling it how to load the plugin.

If you are comfortable with Python virtual environments you can create one now for your project, activate it and run `pip install llm` before the next step.

If you aren't familiar with virtual environments, don't worry: you can develop plugins without them. You'll need to have LLM installed using Homebrew or `pipx` or one of the [other installation options](#).

Installing your plugin to try it out

Having created a directory with a `pyproject.toml` file and an `llm_markov.py` file, you can install your plugin into LLM by running this from inside your `llm-markov` directory:

```
llm install -e .
```

The `-e` stands for “editable” - it means you'll be able to make further changes to the `llm_markov.py` file that will be reflected without you having to reinstall the plugin.

The `.` means the current directory. You can also install editable plugins by passing a path to their directory this:

```
llm install -e path/to/llm-markov
```

To confirm that your plugin has installed correctly, run this command:

```
llm plugins
```

The output should look like this:

```
[
  {
    "name": "llm-markov",
    "hooks": [
      "register_models"
    ],
    "version": "0.1"
  },
  {
    "name": "llm.default_plugins.openai_models",
    "hooks": [
      "register_commands",
      "register_models"
    ]
  }
]
```

This command lists default plugins that are included with LLM as well as new plugins that have been installed.

Now let's try the plugin by running a prompt through it:

```
llm -m markov "the cat sat on the mat"
```

It outputs:

```
hello world
```

Next, we'll make it execute and return the results of a Markov chain.

Building the Markov chain

Markov chains can be thought of as the simplest possible example of a generative language model. They work by building an index of words that have been seen following other words.

Here's what that index looks like for the phrase "the cat sat on the mat"

```
{
  "the": ["cat", "mat"],
  "cat": ["sat"],
  "sat": ["on"],
  "on": ["the"]
}
```

Here's a Python function that builds that data structure from a text input:

```
def build_markov_table(text):
    words = text.split()
    transitions = {}
    # Loop through all but the last word
    for i in range(len(words) - 1):
        word = words[i]
        next_word = words[i + 1]
        transitions.setdefault(word, []).append(next_word)
    return transitions
```

We can try that out by pasting it into the interactive Python interpreter and running this:

```
>>> transitions = build_markov_table("the cat sat on the mat")
>>> transitions
{'the': ['cat', 'mat'], 'cat': ['sat'], 'sat': ['on'], 'on': ['the']}
```

Executing the Markov chain

To execute the model, we start with a word. We look at the options for words that might come next and pick one of those at random. Then we repeat that process until we have produced the desired number of output words.

Some words might not have any following words from our training sentence. For our implementation we will fall back on picking a random word from our collection.

We will implement this as a [Python generator](#), using the `yield` keyword to produce each token:

```
def generate(transitions, length, start_word=None):
    all_words = list(transitions.keys())
    next_word = start_word or random.choice(all_words)
    for i in range(length):
        yield next_word
        options = transitions.get(next_word) or all_words
        next_word = random.choice(options)
```

If you aren't familiar with generators, the above code could also be implemented like this - creating a Python list and returning it at the end of the function:

```
def generate_list(transitions, length, start_word=None):
    all_words = list(transitions.keys())
    next_word = start_word or random.choice(all_words)
    output = []
    for i in range(length):
        output.append(next_word)
        options = transitions.get(next_word) or all_words
        next_word = random.choice(options)
    return output
```

You can try out the generate() function like this:

```
lookup = build_markov_table("the cat sat on the mat")
for word in generate(transitions, 20):
    print(word)
```

Or you can generate a full string sentence with it like this:

```
sentence = " ".join(generate(transitions, 20))
```

Adding that to the plugin

Our execute() method from earlier currently returns the list ["hello world"].

Update that to use our new Markov chain generator instead. Here's the full text of the new llm_markov.py file:

```
import llm
import random

@llm.hookimpl
def register_models(register):
    register(Markov())

def build_markov_table(text):
    words = text.split()
    transitions = {}
    # Loop through all but the last word
    for i in range(len(words) - 1):
        word = words[i]
        next_word = words[i + 1]
        transitions.setdefault(word, []).append(next_word)
    return transitions

def generate(transitions, length, start_word=None):
    all_words = list(transitions.keys())
    next_word = start_word or random.choice(all_words)
    for i in range(length):
        yield next_word
        options = transitions.get(next_word) or all_words
        next_word = random.choice(options)

class Markov(llm.Model):
    model_id = "markov"
```

(continues on next page)

(continued from previous page)

```
def execute(self, prompt, stream, response, conversation):
    text = prompt.prompt
    transitions = build_markov_table(text)
    for word in generate(transitions, 20):
        yield word + ' '
```

The `execute()` method can access the text prompt that the user provided using `prompt.prompt` - `prompt` is a `Prompt` object that might include other more advanced input details as well.

Now when you run this you should see the output of the Markov chain!

```
llm -m markov "the cat sat on the mat"
```

```
the mat the cat sat on the cat sat on the mat cat sat on the mat cat sat on
```

Understanding execute()

The full signature of the `execute()` method is:

```
def execute(self, prompt, stream, response, conversation):
```

The `prompt` argument is a `Prompt` object that contains the text that the user provided, the system prompt and the provided options.

`stream` is a boolean that says if the model is being run in streaming mode.

`response` is the `Response` object that is being created by the model. This is provided so you can write additional information to `response.response_json`, which may be logged to the database.

`conversation` is the `Conversation` that the prompt is a part of - or `None` if no conversation was provided. Some models may use `conversation.responses` to access previous prompts and responses in the conversation and use them to construct a call to the LLM that includes previous context.

Prompts and responses are logged to the database

The prompt and the response will be logged to a SQLite database automatically by LLM. You can see the single most recent addition to the logs using:

```
llm logs -n 1
```

The output should look something like this:

```
[
  {
    "id": "@1h52s4yez2bd1qk2deq49wk8h",
    "model": "markov",
    "prompt": "the cat sat on the mat",
    "system": null,
    "prompt_json": null,
    "options_json": {},
    "response": "on the cat sat on the cat sat on the mat cat sat on the cat sat on the_
↳cat "
```

(continues on next page)

(continued from previous page)

```

"response_json": null,
"conversation_id": "01h52s4yey7zc5rjmczy3ft75g",
"duration_ms": 0,
"datetime_utc": "2023-07-11T15:29:34.685868",
"conversation_name": "the cat sat on the mat",
"conversation_model": "markov"
}
]

```

Plugins can log additional information to the database by assigning a dictionary to the `response.response_json` property during the `execute()` method.

Here's how to include that full transitions table in the `response_json` in the log:

```

def execute(self, prompt, stream, response, conversation):
    text = self.prompt.prompt
    transitions = build_markov_table(text)
    for word in generate(transitions, 20):
        yield word + ' '
    response.response_json = {"transitions": transitions}

```

Now when you run the logs command you'll see that too:

```
llm logs -n 1
```

```

[
  {
    "id": 623,
    "model": "markov",
    "prompt": "the cat sat on the mat",
    "system": null,
    "prompt_json": null,
    "options_json": {},
    "response": "on the mat the cat sat on the cat sat on the mat sat on the cat sat on_
↪the ",
    "response_json": {
      "transitions": {
        "the": [
          "cat",
          "mat"
        ],
        "cat": [
          "sat"
        ],
        "sat": [
          "on"
        ],
        "on": [
          "the"
        ]
      }
    },
    "reply_to_id": null,

```

(continues on next page)

(continued from previous page)

```

"chat_id": null,
"duration_ms": 0,
"datetime_utc": "2023-07-06T01:34:45.376637"
}
]

```

In this particular case this isn't a great idea here though: the `transitions` table is duplicate information, since it can be reproduced from the input data - and it can get really large for longer prompts.

Adding options

LLM models can take options. For large language models these can be things like `temperature` or `top_k`.

Options are passed using the `-o/--option` command line parameters, for example:

```
llm -m gpt4 "ten pet pelican names" -o temperature 1.5
```

We're going to add two options to our Markov chain model:

- `length`: Number of words to generate
- `delay`: a floating point number of Delay in between output token

The `delay` token will let us simulate a streaming language model, where tokens take time to generate and are returned by the `execute()` function as they become ready.

Options are defined using an inner class on the model, called `Options`. It should extend the `llm.Options` class.

First, add this import to the top of your `llm_markov.py` file:

```
from typing import Optional
```

Then add this `Options` class to your model:

```

class Markov(Model):
    model_id = "markov"

    class Options(llm.Options):
        length: Optional[int] = None
        delay: Optional[float] = None

```

Let's add extra validation rules to our options. `Length` must be at least 2. `Duration` must be between 0 and 10.

The `Options` class uses `Pydantic 2`, which can support all sorts of advanced validation rules.

We can also add inline documentation, which can then be displayed by the `llm models --options` command.

Add these imports to the top of `llm_markov.py`:

```
from pydantic import field_validator, Field
```

We can now add `Pydantic` field validators for our two new rules, plus inline documentation:

```

class Options(llm.Options):
    length: Optional[int] = Field(
        description="Number of words to generate",
        default=None

```

(continues on next page)

(continued from previous page)

```

)
delay: Optional[float] = Field(
    description="Seconds to delay between each token",
    default=None
)

@field_validator("length")
def validate_length(cls, length):
    if length is None:
        return None
    if length < 2:
        raise ValueError("length must be >= 2")
    return length

@field_validator("delay")
def validate_delay(cls, delay):
    if delay is None:
        return None
    if not 0 <= delay <= 10:
        raise ValueError("delay must be between 0 and 10")
    return delay

```

Lets test our options validation:

```
llm -m markov "the cat sat on the mat" -o length -1
```

```
Error: length
Value error, length must be >= 2
```

Next, we will modify our execute() method to handle those options. Add this to the beginning of llm_markov.py:

```
import time
```

Then replace the execute() method with this one:

```

def execute(self, prompt, stream, response, conversation):
    text = prompt.prompt
    transitions = build_markov_table(text)
    length = prompt.options.length or 20
    for word in generate(transitions, length):
        yield word + ' '
    if prompt.options.delay:
        time.sleep(prompt.options.delay)

```

Add can_stream = True to the top of the Markov model class, on the line below `model_id = "markov"`. This tells LLM that the model is able to stream content to the console.

The full llm_markov.py file should now look like this:

```

import llm
import random
import time
from typing import Optional

```

(continues on next page)

(continued from previous page)

```
from pydantic import field_validator, Field

@llm.hookimpl
def register_models(register):
    register(Markov())

def build_markov_table(text):
    words = text.split()
    transitions = {}
    # Loop through all but the last word
    for i in range(len(words) - 1):
        word = words[i]
        next_word = words[i + 1]
        transitions.setdefault(word, []).append(next_word)
    return transitions

def generate(transitions, length, start_word=None):
    all_words = list(transitions.keys())
    next_word = start_word or random.choice(all_words)
    for i in range(length):
        yield next_word
        options = transitions.get(next_word) or all_words
        next_word = random.choice(options)

class Markov(llm.Model):
    model_id = "markov"
    can_stream = True

    class Options(llm.Options):
        length: Optional[int] = Field(
            description="Number of words to generate", default=None
        )
        delay: Optional[float] = Field(
            description="Seconds to delay between each token", default=None
        )

    @field_validator("length")
    def validate_length(cls, length):
        if length is None:
            return None
        if length < 2:
            raise ValueError("length must be >= 2")
        return length

    @field_validator("delay")
    def validate_delay(cls, delay):
        if delay is None:
            return None
```

(continues on next page)

(continued from previous page)

```

    if not 0 <= delay <= 10:
        raise ValueError("delay must be between 0 and 10")
    return delay

def execute(self, prompt, stream, response, conversation):
    text = prompt.prompt
    transitions = build_markov_table(text)
    length = prompt.options.length or 20
    for word in generate(transitions, length):
        yield word + " "
        if prompt.options.delay:
            time.sleep(prompt.options.delay)

```

Now we can request a 20 word completion with a 0.1s delay between tokens like this:

```

llm -m markov "the cat sat on the mat" \
-o length 20 -o delay 0.1

```

LLM provides a `--no-stream` option users can use to turn off streaming. Using that option causes LLM to gather the response from the stream and then return it to the console in one block. You can try that like this:

```

llm -m markov "the cat sat on the mat" \
-o length 20 -o delay 0.1 --no-stream

```

In this case it will still delay for 2s total while it gathers the tokens, then output them all at once.

That `--no-stream` option causes the `stream` argument passed to `execute()` to be `false`. Your `execute()` method can then behave differently depending on whether it is streaming or not.

Options are also logged to the database. You can see those here:

```

llm logs -n 1

```

```

[
  {
    "id": 636,
    "model": "markov",
    "prompt": "the cat sat on the mat",
    "system": null,
    "prompt_json": null,
    "options_json": {
      "length": 20,
      "delay": 0.1
    },
    "response": "the mat on the mat on the cat sat on the mat sat on the mat cat sat on_
↪the ",
    "response_json": null,
    "reply_to_id": null,
    "chat_id": null,
    "duration_ms": 2063,
    "datetime_utc": "2023-07-07T03:02:28.232970"
  }
]

```

Distributing your plugin

There are many different options for distributing your new plugin so other people can try it out.

You can create a downloadable wheel or `.zip` or `.tar.gz` files, or share the plugin through GitHub Gists or repositories.

You can also publish your plugin to PyPI, the Python Package Index.

Wheels and sdist packages

The easiest option is to produce a distributable package is to use the `build` command. First, install the `build` package by running this:

```
python -m pip install build
```

Then run `build` in your plugin directory to create the packages:

```
python -m build
```

This will create two files: `dist/llm-markov-0.1.tar.gz` and `dist/llm-markov-0.1-py3-none-any.whl`.

Either of these files can be used to install the plugin:

```
llm install dist/llm_markov-0.1-py3-none-any.whl
```

If you host this file somewhere online other people will be able to install it using `pip install` against the URL to your package:

```
llm install 'https://.../llm_markov-0.1-py3-none-any.whl'
```

You can run the following command at any time to uninstall your plugin, which is useful for testing out different installation methods:

```
llm uninstall llm-markov -y
```

GitHub Gists

A neat quick option for distributing a simple plugin is to host it in a GitHub Gist. These are available for free with a GitHub account, and can be public or private. Gists can contain multiple files but don't support directory structures - which is OK, because our plugin is just two files, `pyproject.toml` and `llm_markov.py`.

Here's an example Gist I created for this tutorial:

<https://gist.github.com/simonw/6e56d48dc2599bffba963cef0db27b6d>

You can turn a Gist into an installable `.zip` URL by right-clicking on the “Download ZIP” button and selecting “Copy Link”. Here's that link for my example Gist:

<https://gist.github.com/simonw/6e56d48dc2599bffba963cef0db27b6d/archive/cc50c854414cb4deab3e3ab17e7e1e07d45cba0c.zip>

The plugin can be installed using the `llm install` command like this:

```
llm install 'https://gist.github.com/simonw/6e56d48dc2599bffba963cef0db27b6d/archive/↪cc50c854414cb4deab3e3ab17e7e1e07d45cba0c.zip'
```

GitHub repositories

The same trick works for regular GitHub repositories as well: the “Download ZIP” button can be found by clicking the green “Code” button at the top of the repository. The URL which that provide scan then be used to install the plugin that lives in that repository.

Publishing plugins to PyPI

The [Python Package Index \(PyPI\)](#) is the official repository for Python packages. You can upload your plugin to PyPI and reserve a name for it - once you have done that, anyone will be able to install your plugin using `llm install <name>`.

Follow [these instructions](#) to publish a package to PyPI. The short version:

```
python -m pip install twine
python -m twine upload dist/*
```

You will need an account on PyPI, then you can enter your username and password - or create a token in the PyPI settings and use `__token__` as the username and the token as the password.

Adding metadata

Before uploading a package to PyPI it’s a good idea to add documentation and expand `pyproject.toml` with additional metadata.

Create a `README.md` file in the root of your plugin directory with instructions about how to install, configure and use your plugin.

You can then replace `pyproject.toml` with something like this:

```
[project]
name = "llm-markov"
version = "0.1"
description = "Plugin for LLM adding a Markov chain generating model"
readme = "README.md"
authors = [{name = "Simon Willison"}]
license = {text = "Apache-2.0"}
classifiers = [
    "License :: OSI Approved :: Apache Software License"
]
dependencies = [
    "llm"
]
requires-python = ">3.7"

[project.urls]
Homepage = "https://github.com/simonw/llm-markov"
Changelog = "https://github.com/simonw/llm-markov/releases"
Issues = "https://github.com/simonw/llm-markov/issues"

[project.entry-points.llm]
markov = "llm_markov"
```

This will pull in your `README` to be displayed as part of your project’s listing page on PyPI.

It adds `llm` as a dependency, ensuring it will be installed if someone tries to install your plugin package without it.

It adds some links to useful pages (you can drop the `project.urls` section if those links are not useful for your project).

You should drop a `LICENSE` file into the GitHub repository for your package as well. I like to use the Apache 2 license [like this](#).

What to do if it breaks

Sometimes you may make a change to your plugin that causes it to break, preventing `llm` from starting. For example you may see an error like this one:

```
$ llm 'hi'
Traceback (most recent call last):
  ...
  File llm-markov/llm_markov.py", line 10
    register(Markov()):
                ^
SyntaxError: invalid syntax
```

You may find that you are unable to uninstall the plugin using `llm uninstall llm-markov` because the command itself fails with the same error.

Should this happen, you can uninstall the plugin after first disabling it using the `LLM_LOAD_PLUGINS` environment variable like this:

```
LLM_LOAD_PLUGINS='' llm uninstall llm-markov
```

2.5.5 Utility functions for plugins

LLM provides some utility functions that may be useful to plugins.

`llm.user_dir()`

LLM stores various pieces of logging and configuration data in a directory on the user's machine.

On macOS this directory is `~/Library/Application Support/io.datasette.llm`, but this will differ on other operating systems.

The `llm.user_dir()` function returns the path to this directory as a `pathlib.Path` object, after creating that directory if it does not yet exist.

Plugins can use this to store their own data in a subdirectory of this directory.

```
import llm
user_dir = llm.user_dir()
plugin_dir = data_path = user_dir / "my-plugin"
plugin_dir.mkdir(exist_ok=True)
data_path = plugin_dir / "plugin-data.db"
```

llm.ModelError

If your model encounters an error that should be reported to the user you can raise this exception. For example:

```
import llm

raise ModelError("MPT model not installed - try running 'llm mpt30b download'")
```

This will be caught by the CLI layer and displayed to the user as an error message.

Response.fake()

When writing tests for a model it can be useful to generate fake response objects, for example in this test from `llm-mpt30b`:

```
def test_build_prompt_conversation():
    model = llm.get_model("mpt")
    conversation = model.conversation()
    conversation.responses = [
        llm.Response.fake(model, "prompt 1", "system 1", "response 1"),
        llm.Response.fake(model, "prompt 2", None, "response 2"),
        llm.Response.fake(model, "prompt 3", None, "response 3"),
    ]
    lines = model.build_prompt(llm.Prompt("prompt 4", model), conversation)
    assert lines == [
        "<|im_start|>system\system 1<|im_end|>\n",
        "<|im_start|>user\nprompt 1<|im_end|>\n",
        "<|im_start|>assistant\nresponse 1<|im_end|>\n",
        "<|im_start|>user\nprompt 2<|im_end|>\n",
        "<|im_start|>assistant\nresponse 2<|im_end|>\n",
        "<|im_start|>user\nprompt 3<|im_end|>\n",
        "<|im_start|>assistant\nresponse 3<|im_end|>\n",
        "<|im_start|>user\nprompt 4<|im_end|>\n",
        "<|im_start|>assistant\n",
    ]
```

The signature of `llm.Response.fake()` is:

```
def fake(cls, model: Model, prompt: str, system: str, response: str):
```

2.6 Model aliases

LLM supports model aliases, which allow you to refer to a model by a short name instead of its full ID.

2.6.1 Listing aliases

To list current aliases, run this:

```
llm aliases
```

Example output:

```
3.5           : gpt-3.5-turbo
chatgpt       : gpt-3.5-turbo
chatgpt-16k   : gpt-3.5-turbo-16k
3.5-16k      : gpt-3.5-turbo-16k
4            : gpt-4
gpt4         : gpt-4
gpt-4-turbo  : gpt-4-1106-preview
4-turbo      : gpt-4-1106-preview
4t          : gpt-4-1106-preview
4-32k       : gpt-4-32k
3.5-instruct : gpt-3.5-turbo-instruct
chatgpt-instruct : gpt-3.5-turbo-instruct
ada         : ada-002 (embedding)
```

Add `--json` to get that list back as JSON:

```
llm aliases list --json
```

Example output:

```
{
  "3.5": "gpt-3.5-turbo",
  "chatgpt": "gpt-3.5-turbo",
  "chatgpt-16k": "gpt-3.5-turbo-16k",
  "3.5-16k": "gpt-3.5-turbo-16k",
  "4": "gpt-4",
  "gpt4": "gpt-4",
  "4-32k": "gpt-4-32k",
  "ada": "ada-002"
}
```

2.6.2 Adding a new alias

The `llm aliases set <alias> <model-id>` command can be used to add a new alias:

```
llm aliases set turbo gpt-3.5-turbo-16k
```

Now you can run the `gpt-3.5-turbo-16k` model using the `turbo` alias like this:

```
llm -m turbo 'An epic Greek-style saga about a cheesecake that builds a SQL database_
↳from scratch'
```

Aliases can be set for both regular models and *embedding models* using the same command. To set an alias of oai for the OpenAI `ada-002` embedding model use this:

```
llm aliases set oai ada-002
```

Now you can embed a string using that model like so:

```
llm embed -c 'hello world' -m oai
```

Output:

```
[-0.014945968054234982, 0.0014304015785455704, ...]
```

2.6.3 Removing an alias

The `llm aliases remove <alias>` command will remove the specified alias:

```
llm aliases remove turbo
```

2.6.4 Viewing the aliases file

Aliases are stored in an `aliases.json` file in the LLM configuration directory.

To see the path to that file, run this:

```
llm aliases path
```

To view the content of that file, run this:

```
cat "$(llm aliases path)"
```

2.7 Python API

LLM provides a Python API for executing prompts, in addition to the command-line interface.

Understanding this API is also important for writing *Plugins*.

2.7.1 Basic prompt execution

To run a prompt against the `gpt-3.5-turbo` model, run this:

```
import llm

model = llm.get_model("gpt-3.5-turbo")
model.key = 'YOUR_API_KEY_HERE'
response = model.prompt("Five surprising names for a pet pelican")
print(response.text())
```

The `llm.get_model()` function accepts model names or aliases - so `chatgpt` would work here too.

The `__str__()` method of `response` also returns the text of the response, so you can do this instead:

```
print(response)
```

You can run this command to see a list of available models and their aliases:

```
llm models
```

If you have set a `OPENAI_API_KEY` environment variable you can omit the `model.key =` line.

Calling `llm.get_model()` with an invalid model name will raise a `llm.UnknownModelError` exception.

System prompts

For models that accept a system prompt, pass it as `system="..."`:

```
response = model.prompt(  
    "Five surprising names for a pet pelican",  
    system="Answer like GLaDOS"  
)
```

Model options

For models that support options (view those with `llm models --options`) you can pass options as keyword arguments to the `.prompt()` method:

```
model = llm.get_model("gpt-3.5-turbo")  
model.key = "... key here ..."  
print(model.prompt("Names for otters", temperature=0.2))
```

Models from plugins

Any models you have installed as plugins will also be available through this mechanism, for example to use Google's PaLM 2 model with `llm-palm`

```
pip install llm-palm
```

```
import llm  
  
model = llm.get_model("palm")  
model.key = 'YOUR_API_KEY_HERE'  
response = model.prompt("Five surprising names for a pet pelican")  
print(response.text())
```

You can omit the `model.key =` line for models that do not use an API key

2.7.2 Streaming responses

For models that support it you can stream responses as they are generated, like this:

```
response = model.prompt("Five diabolical names for a pet goat")
for chunk in response:
    print(chunk, end="")
```

The `response.text()` method described earlier does this for you - it runs through the iterator and gathers the results into a string.

If a response has been evaluated, `response.text()` will continue to return the same string.

2.7.3 Conversations

LLM supports *conversations*, where you ask follow-up questions of a model as part of an ongoing conversation.

To start a new conversation, use the `model.conversation()` method:

```
model = llm.get_model("gpt-3.5-turbo")
model.key = 'YOUR_API_KEY_HERE'
conversation = model.conversation()
```

You can then use the `conversation.prompt()` method to execute prompts against this conversation:

```
response = conversation.prompt("Five fun facts about pelicans")
print(response.text())
```

This works exactly the same as the `model.prompt()` method, except that the conversation will be maintained across multiple prompts. So if you run this next:

```
response2 = conversation.prompt("Now do skunks")
print(response2.text())
```

You will get back five fun facts about skunks.

Access `conversation.responses` for a list of all of the responses that have so far been returned during the conversation.

2.7.4 Other functions

The `llm` top level package includes some useful utility functions.

`set_alias(alias, model_id)`

The `llm.set_alias()` function can be used to define a new alias:

```
import llm

llm.set_alias("turbo", "gpt-3.5-turbo")
```

The second argument can be a model identifier or another alias, in which case that alias will be resolved.

If the `aliases.json` file does not exist or contains invalid JSON it will be created or overwritten.

remove_alias(alias)

Removes the alias with the given name from the `aliases.json` file.

Raises `KeyError` if the alias does not exist.

```
import llm  
llm.remove_alias("turbo")
```

2.8 Prompt templates

Prompt templates can be created to reuse useful prompts with different input data.

2.8.1 Getting started

The easiest way to create a template is using the `--save template_name` option.

Here's how to create a template for summarizing text:

```
llm 'Summarize this: $input' --save summarize
```

You can also create templates using system prompts:

```
llm --system 'Summarize this' --save summarize
```

You can set the default model for a template using `--model`:

```
llm --system 'Summarize this' --model gpt-4 --save summarize
```

You can also save default parameters:

```
llm --system 'Summarize this text in the voice of $voice' \  
--model gpt-4 -p voice GladOS --save summarize
```

2.8.2 Using a template

You can execute a named template using the `-t/--template` option:

```
curl -s https://example.com/ | llm -t summarize
```

This can be combined with the `-m` option to specify a different model:

```
curl -s https://llm.datasette.io/en/latest/ | \  
llm -t summarize -m gpt-3.5-turbo-16k
```

2.8.3 Listing available templates

This command lists all available templates:

```
llm templates
```

The output looks something like this:

```
cmd      : system: reply with macos terminal commands only, no extra information
glados   : system: You are GladOS prompt: Summarize this: $input
```

2.8.4 Templates as YAML files

Templates are stored as YAML files on disk.

You can edit (or create) a YAML file for a template using the `llm templates edit` command:

```
llm templates edit summarize
```

This will open the system default editor.

Tip: You can control which editor will be used here using the `EDITOR` environment variable - for example, to use VS Code:

```
export EDITOR="code -w"
```

Add that to your `~/ .zshrc` or `~/ .bashrc` file depending on which shell you use (zsh is the default on macOS since macOS Catalina in 2019).

You can also create a file called `summary.yaml` in the folder shown by running `llm templates path`, for example:

```
$ llm templates path
/Users/simon/Library/Application Support/io.datasette.llm/templates
```

You can also represent this template as a YAML dictionary with a `prompt:` key, like this one:

```
prompt: 'Summarize this: $input'
```

Or use YAML multi-line strings for longer inputs. I created this using `llm templates edit steampunk`:

```
prompt: >
  Summarize the following text.

  Insert frequent satirical steampunk-themed illustrative anecdotes.
  Really go wild with that.

  Text to summarize: $input
```

The `prompt: >` causes the following indented text to be treated as a single string, with newlines collapsed to spaces. Use `prompt: |` to preserve newlines.

Running that with `llm -t steampunk` against GPT-4 (via `strip-tags` to remove HTML tags from the input and minify whitespace):

```
curl -s 'https://til.simonwillison.net/macOS/imovie-slides-and-audio' | \
strip-tags -m | llm -t steampunk -m 4
```

Output:

In a fantastical steampunk world, Simon Willison decided to merge an old MP3 recording with slides from the talk using iMovie. After exporting the slides as images and importing them into iMovie, he had to disable the default Ken Burns effect using the “Crop” tool. Then, Simon manually synchronized the audio by adjusting the duration of each image. Finally, he published the masterpiece to YouTube, with the whimsical magic of steampunk-infused illustrations leaving his viewers in awe.

2.8.5 System templates

When working with models that support system prompts (such as `gpt-3.5-turbo` and `gpt-4`) you can set a system prompt using a `system:` key like so:

```
system: Summarize this
```

If you specify only a system prompt you don't need to use the `$input` variable - `llm` will use the user's input as the whole of the regular prompt, which will then be processed using the instructions set in that system prompt.

You can combine system and regular prompts like so:

```
system: You speak like an excitable Victorian adventurer
prompt: 'Summarize this: $input'
```

2.8.6 Additional template variables

Templates that work against the user's normal input (content that is either piped to the tool via standard input or passed as a command-line argument) use just the `$input` variable.

You can use additional named variables. These will then need to be provided using the `-p/--param` option when executing the template.

Here's an example template called `recipe`, created using `llm templates edit recipe`:

```
prompt: |
Suggest a recipe using ingredients: $ingredients

It should be based on cuisine from this country: $country
```

This can be executed like so:

```
llm -t recipe -p ingredients 'sausages, milk' -p country Germany
```

My output started like this:

Recipe: German Sausage and Potato Soup

Ingredients:

- 4 German sausages
- 2 cups whole milk

This example combines input piped to the tool with additional parameters. Call this `summarize`:

```
system: Summarize this text in the voice of $voice
```

Then to run it:

```
curl -s 'https://til.simonwillison.net/macOS/imovie-slides-and-audio' | \
strip-tags -m | llm -t summarize -p voice GladOS
```

I got this:

My previous test subject seemed to have learned something new about iMovie. They exported keynote slides as individual images [...] Quite impressive for a human.

2.8.7 Specifying default parameters

You can also specify default values for parameters, using a `defaults:` key.

```
system: Summarize this text in the voice of $voice
defaults:
  voice: GladOS
```

When running without `-p` it will choose the default:

```
curl -s 'https://til.simonwillison.net/macOS/imovie-slides-and-audio' | \
strip-tags -m | llm -t summarize
```

But you can override the defaults with `-p`:

```
curl -s 'https://til.simonwillison.net/macOS/imovie-slides-and-audio' | \
strip-tags -m | llm -t summarize -p voice Yoda
```

I got this:

Text, summarize in Yoda's voice, I will: "Hmm, young padawan. Summary of this text, you seek. Hmmm.
...

2.8.8 Setting a default model for a template

Templates executed using `llm -t template-name` will execute using the default model that the user has configured for the tool - or `gpt-3.5-turbo` if they have not configured their own default.

You can specify a new default model for a template using the `model:` key in the associated YAML. Here's a template called `roast`:

```
model: gpt-4
system: roast the user at every possible opportunity, be succinct
```

Example:

```
llm -t roast 'How are you today?'
```

I'm doing great but with your boring questions, I must admit, I've seen more life in a cemetery.

2.9 Logging to SQLite

llm defaults to logging all prompts and responses to a SQLite database.

You can find the location of that database using the `llm logs path` command:

```
llm logs path
```

On my Mac that outputs:

```
/Users/simon/Library/Application Support/io.datasette.llm/logs.db
```

This will differ for other operating systems.

To avoid logging an individual prompt, pass `--no-log` or `-n` to the command:

```
llm 'Ten names for cheesecakes' -n
```

To turn logging by default off:

```
llm logs off
```

If you've turned off logging you can still log an individual prompt and response by adding `--log`:

```
llm 'Five ambitious names for a pet pterodactyl' --log
```

To turn logging by default back on again:

```
llm logs on
```

To see the status of the logs database, run this:

```
llm logs status
```

Example output:

```
Logging is ON for all prompts
Found log database at /Users/simon/Library/Application Support/io.datasette.llm/logs.db
Number of conversations logged: 33
Number of responses logged:    48
Database file size:           19.96MB
```

2.9.1 Viewing the logs

You can view the logs using the `llm logs` command:

```
llm logs
```

This will output the three most recent logged items in Markdown format

Add `--json` to get the log messages in JSON instead:

```
llm logs --json
```

Add `-n 10` to see the ten most recent items:

```
llm logs -n 10
```

Or `-n 0` to see everything that has ever been logged:

```
llm logs -n 0
```

You can truncate the display of the prompts and responses using the `-t/--truncate` option. This can help make the JSON output more readable:

```
llm logs -n 5 -t --json
```

Logs for a conversation

To view the logs for the most recent *conversation* you have had with a model, use `-c`:

```
llm logs -c
```

To see logs for a specific conversation based on its ID, use `--cid ID` or `--conversation ID`:

```
llm logs --cid 01h82n0q9crqtnzmf13gkyxawg
```

Searching the logs

You can search the logs for a search term in the `prompt` or the `response` columns.

```
llm logs -q 'cheesecake'
```

The most relevant terms will be shown at the bottom of the output.

Filtering by model

You can filter to logs just for a specific model (or model alias) using `-m/--model`:

```
llm logs -m chatgpt
```

Browsing logs using Datasette

You can also use [Datasette](#) to browse your logs like this:

```
datasette "$(llm logs path)"
```

2.9.2 SQL schema

Here's the SQL schema used by the `logs.db` database:

```
CREATE TABLE [conversations] (  
  [id] TEXT PRIMARY KEY,  
  [name] TEXT,  
  [model] TEXT  
);  
CREATE TABLE [responses] (  
  [id] TEXT PRIMARY KEY,  
  [model] TEXT,  
  [prompt] TEXT,  
  [system] TEXT,  
  [prompt_json] TEXT,  
  [options_json] TEXT,  
  [response] TEXT,  
  [response_json] TEXT,  
  [conversation_id] TEXT REFERENCES [conversations]([id]),  
  [duration_ms] INTEGER,  
  [datetime_utc] TEXT  
);  
CREATE VIRTUAL TABLE [responses_fts] USING FTS5 (  
  [prompt],  
  [response],  
  content=[responses]  
);
```

`responses_fts` configures SQLite full-text search against the `prompt` and `response` columns in the `responses` table.

2.10 Related tools

The following tools are designed to be used with LLM:

2.10.1 strip-tags

`strip-tags` is a command for stripping tags from HTML. This is useful when working with LLMs because HTML tags can use up a lot of your token budget.

Here's how to summarize the front page of the New York Times, by both stripping tags and filtering to just the elements with `class="story-wrapper"`:

```
curl -s https://www.nytimes.com/ \  
  | strip-tags .story-wrapper \  
  | llm -s 'summarize the news'
```

`llm`, `ttok` and `strip-tags`—CLI tools for working with ChatGPT and other LLMs describes ways to use `strip-tags` in more detail.

2.10.2 ttok

`ttok` is a command-line tool for counting OpenAI tokens. You can use it to check if input is likely to fit in the token limit for GPT 3.5 or GPT4:

```
cat my-file.txt | ttok
```

```
125
```

It can also truncate input down to a desired number of tokens:

```
ttok This is too many tokens -t 3
```

```
This is too
```

This is useful for truncating a large document down to a size where it can be processed by an LLM.

2.10.3 Symbex

`Symbex` is a tool for searching for symbols in Python codebases. It's useful for extracting just the code for a specific problem and then piping that into LLM for explanation, refactoring or other tasks.

Here's how to use it to find all functions that match `test*csv*` and use those to guess what the software under test does:

```
symbex 'test*csv*' | \
  llm --system 'based on these tests guess what this tool does'
```

It can also be used to export symbols in a format that can be piped to `llm embed-multi` in order to create embeddings:

```
symbex '*' '*:*' --nl | \
  llm embed-multi symbols - \
  --format nl --database embeddings.db --store
```

For more examples see [Symbex: search Python code for functions and classes, then pipe them into a LLM](#).

2.11 CLI reference

This page lists the `--help` output for all of the `llm` commands.

2.11.1 llm --help

```
Usage: llm [OPTIONS] COMMAND [ARGS]...

Access large language models from the command-line

Documentation: https://llm.datasette.io/

To get started, obtain an OpenAI key and set it like this:
```

(continues on next page)

(continued from previous page)

```
$ llm keys set openai
Enter key: ...
```

Then execute a prompt like this:

```
llm 'Five outrageous names for a pet pelican'
```

Options:

```
--version Show the version and exit.
--help    Show this message and exit.
```

Commands:

```
prompt*    Execute a prompt
aliases    Manage model aliases
chat       Hold an ongoing chat with a model.
collections View and manage collections of embeddings
embed      Embed text and store or return the result
embed-models Manage available embedding models
embed-multi Store embeddings for multiple strings at once
install    Install packages from PyPI into the same environment as LLM
keys       Manage stored API keys for different models
logs       Tools for exploring logged prompts and responses
models     Manage available models
openai     Commands for working directly with the OpenAI API
plugins    List installed plugins
similar    Return top N similar IDs from a collection
templates  Manage stored prompt templates
uninstall  Uninstall Python packages from the LLM environment
```

llm prompt -help

```
Usage: llm prompt [OPTIONS] [PROMPT]
```

Execute a prompt

Documentation: <https://llm.datasette.io/en/stable/usage.html>

Options:

```
-s, --system TEXT      System prompt to use
-m, --model TEXT       Model to use
-o, --option <TEXT TEXT>... key/value options for the model
-t, --template TEXT    Template to use
-p, --param <TEXT TEXT>... Parameters for template
--no-stream            Do not stream output
-n, --no-log           Don't log to database
--log                 Log prompt and response to the database
-c, --continue        Continue the most recent conversation.
--cid, --conversation TEXT Continue the conversation with the given ID.
--key TEXT            API key to use
--save TEXT           Save prompt with this template name
--help                Show this message and exit.
```

llm chat --help

Usage: llm chat [OPTIONS]

Hold an ongoing chat **with** a model.

Options:

-s, --system TEXT	System prompt to use
-m, --model TEXT	Model to use
-c, --continue	Continue the most recent conversation.
--cid, --conversation TEXT	Continue the conversation with the given ID.
-t, --template TEXT	Template to use
-p, --param <TEXT TEXT>...	Parameters for template
-o, --option <TEXT TEXT>...	key/value options for the model
--no-stream	Do not stream output
--key TEXT	API key to use
--help	Show this message and exit.

llm keys --help

Usage: llm keys [OPTIONS] COMMAND [ARGS]...

Manage stored API keys **for** different models

Options:

--help Show this message **and** exit.

Commands:

list*	List names of all stored keys
path	Output the path to the keys.json file
set	Save a key in the keys.json file

llm keys list --help

Usage: llm keys list [OPTIONS]

List names of **all** stored keys

Options:

--help Show this message **and** exit.

llm keys path –help

Usage: llm keys path [OPTIONS]

Output the path to the keys.json file

Options:

--help Show this message **and** exit.

llm keys set –help

Usage: llm keys set [OPTIONS] NAME

Save a key in the keys.json file

Example usage:

```
$ llm keys set openai
Enter key: ...
```

Options:

--value TEXT Value to set
--help Show this message and exit.

llm logs –help

Usage: llm logs [OPTIONS] COMMAND [ARGS]...

Tools **for** exploring logged prompts **and** responses

Options:

--help Show this message **and** exit.

Commands:

list* Show recent logged prompts **and** their responses
off Turn off logging **for all** prompts
on Turn on logging **for all** prompts
path Output the path to the logs.db file
status Show current status of database logging

llm logs path –help

Usage: llm logs path [OPTIONS]

Output the path to the logs.db file

Options:

--help Show this message **and** exit.

llm logs status –help

Usage: llm logs status [OPTIONS]

Show current status of database logging

Options:

--help Show this message **and** exit.

llm logs on –help

Usage: llm logs on [OPTIONS]

Turn on logging **for all** prompts

Options:

--help Show this message **and** exit.

llm logs off –help

Usage: llm logs off [OPTIONS]

Turn off logging **for all** prompts

Options:

--help Show this message **and** exit.

llm logs list –help

Usage: llm logs list [OPTIONS]

Show recent logged prompts **and** their responses

Options:

-n, --count INTEGER Number of entries to show - defaults to 3, use 0
for all

(continues on next page)

(continued from previous page)

```

-p, --path FILE          Path to log database
-m, --model TEXT         Filter by model or model alias
-q, --query TEXT         Search for logs matching this string
-t, --truncate           Truncate long strings in output
-c, --current            Show logs from the current conversation
--cid, --conversation TEXT Show logs for this conversation ID
--json                   Output logs as JSON
--help                   Show this message and exit.

```

llm models -help

```
Usage: llm models [OPTIONS] COMMAND [ARGS]...
```

Manage available models

Options:

```
--help Show this message and exit.
```

Commands:

```
list* List available models
default Show or set the default model
```

llm models list -help

```
Usage: llm models list [OPTIONS]
```

List available models

Options:

```
--options Show options for each model, if available
--help Show this message and exit.
```

llm models default -help

```
Usage: llm models default [OPTIONS] [MODEL]
```

Show **or set** the default model

Options:

```
--help Show this message and exit.
```

llm templates –help

```
Usage: llm templates [OPTIONS] COMMAND [ARGS]...
```

Manage stored prompt templates

Options:

--help Show this message and exit.

Commands:

list* List available prompt templates
edit Edit the specified prompt template using the default \$EDITOR
path Output the path to the templates directory
show Show the specified prompt template

llm templates list –help

```
Usage: llm templates list [OPTIONS]
```

List available prompt templates

Options:

--help Show this message and exit.

llm templates show –help

```
Usage: llm templates show [OPTIONS] NAME
```

Show the specified prompt template

Options:

--help Show this message and exit.

llm templates edit –help

```
Usage: llm templates edit [OPTIONS] NAME
```

Edit the specified prompt template using the default \$EDITOR

Options:

--help Show this message and exit.

llm templates path –help

Usage: llm templates path [OPTIONS]

Output the path to the templates directory

Options:

--help Show this message **and** exit.

llm aliases –help

Usage: llm aliases [OPTIONS] COMMAND [ARGS]...

Manage model aliases

Options:

--help Show this message **and** exit.

Commands:

list* List current aliases
path Output the path to the aliases.json file
remove Remove an alias
set Set an alias **for** a model

llm aliases list –help

Usage: llm aliases **list** [OPTIONS]

List current aliases

Options:

--json Output **as** JSON
--help Show this message **and** exit.

llm aliases set –help

Usage: llm aliases set [OPTIONS] ALIAS MODEL_ID

Set an alias for a model

Example usage:

```
$ llm aliases set turbo gpt-3.5-turbo
```

Options:

--help Show this message **and** exit.

llm aliases remove –help

Usage: llm aliases remove [OPTIONS] ALIAS

Remove an alias

Example usage:

```
$ llm aliases remove turbo
```

Options:

--help Show this message and exit.

llm aliases path –help

Usage: llm aliases path [OPTIONS]

Output the path to the aliases.json file

Options:

--help Show this message **and** exit.

llm plugins –help

Usage: llm plugins [OPTIONS]

List installed plugins

Options:

--all Include built-**in** default plugins

--help Show this message **and** exit.

llm install –help

Usage: llm install [OPTIONS] [PACKAGES]...

Install packages **from** PyPI into the same environment **as** LLM

Options:

-U, --upgrade Upgrade packages to latest version

-e, --editable TEXT Install a project **in** editable mode **from** **this** path

--force-reinstall Reinstall **all** packages even **if** they are already up-to-date

--no-cache-dir Disable the cache

--help Show this message **and** exit.

llm uninstall -help

Usage: llm uninstall [OPTIONS] PACKAGES...

Uninstall Python packages **from the** LLM environment

Options:

-y, --yes Don't ask for confirmation
--help Show this message **and** exit.

llm embed -help

Usage: llm embed [OPTIONS] [COLLECTION] [ID]

Embed text **and** store **or return** the result

Options:

-i, --input PATH File to embed
-m, --model TEXT Embedding model to use
--store Store the text itself **in** the database
-d, --database FILE
-c, --content TEXT Content to embed
--binary Treat **input as** binary data
--metadata TEXT JSON **object** metadata to store
-f, --format [json|blob|base64|hex] Output **format**
--help Show this message **and** exit.

llm embed-multi -help

Usage: llm embed-multi [OPTIONS] COLLECTION [INPUT_PATH]

Store embeddings **for** multiple strings at once

Input can be CSV, TSV **or** a JSON **list** of objects.

The first column **is** treated **as** an ID - **all** other columns are assumed to be text that should be concatenated together **in** order to calculate the embeddings.

Input data can come **from one** of three sources:

1. A CSV, JSON, TSV **or** JSON-nl file (including on standard **input**)
2. A SQL query against a SQLite database
3. A directory of files

Options:

--format [json|csv|tsv|nl] Format of **input** file - defaults to auto-detect
--files <DIRECTORY TEXT>... Embed files **in** this directory - specify directory **and** glob pattern

(continues on next page)

(continued from previous page)

```

--encoding TEXT          Encoding to use when reading --files
--binary                Treat --files as binary data
--sql TEXT              Read input using this SQL query
--attach <TEXT FILE>... Additional databases to attach - specify alias
                        and file path
--batch-size INTEGER    Batch size to use when running embeddings
--prefix TEXT           Prefix to add to the IDs
-m, --model TEXT        Embedding model to use
--store                 Store the text itself in the database
-d, --database FILE     Show this message and exit.
--help

```

llm similar -help

```
Usage: llm similar [OPTIONS] COLLECTION [ID]
```

Return top N similar IDs from a collection

Example usage:

```
llm similar my-collection -c "I like cats"
```

Or to find content similar to a specific stored ID:

```
llm similar my-collection 1234
```

Options:

```

-i, --input PATH        File to embed for comparison
-c, --content TEXT      Content to embed for comparison
--binary                Treat input as binary data
-n, --number INTEGER    Number of results to return
-d, --database FILE     Show this message and exit.
--help

```

llm embed-models -help

```
Usage: llm embed-models [OPTIONS] COMMAND [ARGS]...
```

Manage available embedding models

Options:

```
--help Show this message and exit.
```

Commands:

```

list* List available embedding models
default Show or set the default embedding model

```

llm embed-models list –help

Usage: llm embed-models list [OPTIONS]

List available embedding models

Options:

--help Show this message and exit.

llm embed-models default –help

Usage: llm embed-models default [OPTIONS] [MODEL]

Show or set the default embedding model

Options:

--remove-default Reset to specifying no default model

--help Show this message and exit.

llm collections –help

Usage: llm collections [OPTIONS] COMMAND [ARGS]...

View and manage collections of embeddings

Options:

--help Show this message and exit.

Commands:

list* View a list of collections

delete Delete the specified collection

path Output the path to the embeddings database

llm collections path –help

Usage: llm collections path [OPTIONS]

Output the path to the embeddings database

Options:

--help Show this message and exit.

llm collections list –help

Usage: llm collections list [OPTIONS]

View a list of collections

Options:

-d, --database FILE Path to embeddings database
--json Output as JSON
--help Show this message and exit.

llm collections delete –help

Usage: llm collections delete [OPTIONS] COLLECTION

Delete the specified collection

Example usage:

```
llm collections delete my-collection
```

Options:

-d, --database FILE Path to embeddings database
--help Show this message and exit.

llm openai –help

Usage: llm openai [OPTIONS] COMMAND [ARGS]...

Commands for working directly with the OpenAI API

Options:

--help Show this message and exit.

Commands:

models List models available to you from the OpenAI API

llm openai models –help

Usage: llm openai models [OPTIONS]

List models available to you from the OpenAI API

Options:

--json Output as JSON
--key TEXT OpenAI API key
--help Show this message and exit.

2.12 Contributing

To contribute to this tool, first checkout the code. Then create a new virtual environment:

```
cd llm
python -m venv venv
source venv/bin/activate
```

Or if you are using pipenv:

```
pipenv shell
```

Now install the dependencies and test dependencies:

```
pip install -e '[test]'
```

To run the tests:

```
pytest
```

2.12.1 Debugging tricks

The default OpenAI plugin has a debugging mechanism for showing the exact responses that came back from the OpenAI API.

Set the `LLM_OPENAI_SHOW_RESPONSES` environment variable like this:

```
LLM_OPENAI_SHOW_RESPONSES=1 llm -m chatgpt 'three word slogan for an an otter-run bakery'
```

This will output the response (including streaming responses) to standard error, as shown in [issues 286](#).

2.12.2 Documentation

Documentation for this project uses [MyST](#) - it is written in Markdown and rendered using Sphinx.

To build the documentation locally, run the following:

```
cd docs
pip install -r requirements.txt
make livehtml
```

This will start a live preview server, using [sphinx-autobuild](#).

The CLI `--help` examples in the documentation are managed using [Cog](#). Update those files like this:

```
just cog
```

You'll need [Just](#) installed to run this command.

2.12.3 Release process

To release a new version:

1. Update docs/changelog.md with the new changes.
2. Update the version number in setup.py
3. Create a [GitHub release](#) for the new version.
4. Wait for the package to push to PyPI and then...
5. Run the [regenerate.yaml](#) workflow to update the Homebrew tap to the latest version.

2.13 Changelog

2.13.1 0.12 (2023-11-06)

- Support for the new [GPT-4 Turbo model](#) from OpenAI. Try it using `llm chat -m gpt-4-turbo` or `llm chat -m 4t`. [#323](#)
- New `-o seed 1` option for OpenAI models which sets a seed that can attempt to evaluate the prompt deterministically. [#324](#)

2.13.2 0.11.2 (2023-11-06)

- Pin to version of OpenAI Python library prior to 1.0 to avoid breaking. [#327](#)

2.13.3 0.11.1 (2023-10-31)

- Fixed a bug where `llm embed -c "text"` did not correctly pick up the configured *default embedding model*. [#317](#)
- New plugins: `llm-python`, `llm-bedrock-anthropic` and `llm-embed-jina` (described in [Execute Jina embeddings with a CLI using llm-embed-jina](#)).
- `llm-gpt4all` now uses the new GGUF model format. [simonw/llm-gpt4all#16](#)

2.13.4 0.11 (2023-09-18)

LLM now supports the new OpenAI `gpt-3.5-turbo-instruct` model, and OpenAI completion (as opposed to chat completion) models in general. [#284](#)

```
llm -m gpt-3.5-turbo-instruct 'Reasons to tame a wild beaver:'
```

OpenAI completion models like this support a `-o logprobs 3` option, which accepts a number between 1 and 5 and will include the log probabilities (for each produced token, what were the top 3 options considered by the model) in the logged response.

```
llm -m gpt-3.5-turbo-instruct 'Say hello succinctly' -o logprobs 3
```

You can then view the logprobs that were recorded in the SQLite logs database like this:

```
sqlite-utils "$(llm logs path)" \  
'select * from responses order by id desc limit 1' | \  
jq '[0].response_json' -r | jq
```

Truncated output looks like this:

```
[  
  {  
    "text": "Hi",  
    "top_logprobs": [  
      {  
        "Hi": -0.13706253,  
        "Hello": -2.3714375,  
        "Hey": -3.3714373  
      }  
    ]  
  },  
  {  
    "text": " there",  
    "top_logprobs": [  
      {  
        " there": -0.96057636,  
        "!\"": -0.5855763,  
        ".\"": -3.2574513  
      }  
    ]  
  }  
]
```

Also in this release:

- The `llm.user_dir()` function, used by plugins, now ensures the directory exists before returning it. #275
- New `LLM_OPENAI_SHOW_RESPONSES=1` environment variable for displaying the full HTTP response returned by OpenAI compatible APIs. #286
- The `llm embed-multi` command now has a `--batch-size X` option for setting the batch size to use when processing embeddings - useful if you have limited memory available. #273
- The `collection.embed_multi()` method also now accepts an optional `batch_size=int` argument.
- Fixed two bugs with `llm embed-multi --files` relating to handling of directories. Thanks, [ealvar3z](#). #274, #280

2.13.5 0.10 (2023-09-12)

The two major features in this release are the *llm chat command* and support for **embedding binary data**.

See [Build an image search engine with llm-clip](#), [chat with models with llm chat](#) for more background on these features.

llm chat

The new `llm chat` command starts an ongoing chat conversation with a model in your terminal. It works with all models supported by LLM and its *plugins*, including locally installed models such as Llama 2. #231

This offers a big performance boost for local models, since they don't need to be freshly loaded into memory for each prompt.

Here's an example chat with Llama 2 13B, provided by the `llm-mlc` plugin.

```
llm chat -m mlc-chat-Llama-2-13b-chat-hf-q4f16_1
```

This starts a chat session:

```
Type 'exit' or 'quit' to exit
Type '!multi' to enter multiple lines, then '!end' to finish
> Who are you?
Hello! I'm just an AI, here to assist you with any questions you may have.
My name is LLaMA, and I'm a large language model trained to provide helpful
and informative responses to a wide range of questions and topics. I'm here
to help you find the information you need, while ensuring a safe and
respectful conversation. Please feel free to ask me anything, and I'll do my
best to provide a helpful and accurate response.
> Tell me a joke about otters
Sure, here's a joke about otters:

Why did the otter go to the party?

Because he heard it was a "whale" of a time!

(Get it? Whale, like a big sea mammal, but also a "wild" or "fun" time.
Otters are known for their playful and social nature, so it's a lighthearted
and silly joke.)

I hope that brought a smile to your face! Do you have any other questions or
topics you'd like to discuss?
> exit
```

Chat sessions are *logged to SQLite* - use `llm logs` to view them. They can accept system prompts, templates and model options - consult *the chat documentation* for details.

Binary embedding support

LLM's *embeddings feature* has been expanded to provide support for embedding binary data, in addition to text. #254

This enables models like `CLIP`, supported by the new `llm-clip` plugin.

`CLIP` is a multi-modal embedding model which can embed images and text into the same vector space. This means you can use it to create an embedding index of photos, and then search for the embedding vector for "a happy dog" and get back images that are semantically closest to that string.

To create embeddings for every JPEG in a directory stored in a photos collection, run:

```
llm install llm-clip
llm embed-multi photos --files photos/ '*.jpg' --binary -m clip
```

Now you can search for photos of racoons using:

```
llm similar photos -c 'raccoon'
```

This spits out a list of images, ranked by how similar they are to the string “raccoon”:

```
{"id": "IMG_4801.jpeg", "score": 0.28125139257127457, "content": null, "metadata": null}
{"id": "IMG_4656.jpeg", "score": 0.26626441704164294, "content": null, "metadata": null}
{"id": "IMG_2944.jpeg", "score": 0.2647445926996852, "content": null, "metadata": null}
...
```

Also in this release

- The `LLM_LOAD_PLUGINS` environment variable can be used to control which plugins are loaded when `llm` starts running. #256
- The `llm plugins --all` option includes builtin plugins in the list of plugins. #259
- The `llm embed-db` family of commands has been renamed to `llm collections`. #229
- `llm embed-multi --files` now has an `--encoding` option and defaults to falling back to `latin-1` if a file cannot be processed as `utf-8`. #225

2.13.6 0.10a1 (2023-09-11)

- Support for embedding binary data. #254
- `llm chat` now works for models with API keys. #247
- `llm chat -o` for passing options to a model. #244
- `llm chat --no-stream` option. #248
- `LLM_LOAD_PLUGINS` environment variable. #256
- `llm plugins --all` option for including builtin plugins. #259
- `llm embed-db` has been renamed to `llm collections`. #229
- Fixed bug where `llm embed -c` option was treated as a filepath, not a string. Thanks, `mhalle`. #263

2.13.7 0.10a0 (2023-09-04)

- New `llm chat` command for starting an interactive terminal chat with a model. #231
- `llm embed-multi --files` now has an `--encoding` option and defaults to falling back to `latin-1` if a file cannot be processed as `utf-8`. #225

2.13.8 0.9 (2023-09-03)

The big new feature in this release is support for **embeddings**. See [LLM now provides tools for working with embeddings](#) for additional details.

Embedding models take a piece of text - a word, sentence, paragraph or even a whole article, and convert that into an array of floating point numbers. [#185](#)

This embedding vector can be thought of as representing a position in many-dimensional-space, where the distance between two vectors represents how semantically similar they are to each other within the content of a language model.

Embeddings can be used to find **related documents**, and also to implement **semantic search** - where a user can search for a phrase and get back results that are semantically similar to that phrase even if they do not share any exact keywords.

LLM now provides both CLI and Python APIs for working with embeddings. Embedding models are defined by plugins, so you can install additional models using the *plugins mechanism*.

The first two embedding models supported by LLM are:

- OpenAI's *ada-002* embedding model, available via an inexpensive API if you set an OpenAI key using `llm keys set openai`.
- The *sentence-transformers* family of models, available via the new `llm-sentence-transformers` plugin.

See [Embedding with the CLI](#) for detailed instructions on working with embeddings using LLM.

The new commands for working with embeddings are:

- *llm embed* - calculate embeddings for content and return them to the console or store them in a SQLite database.
- *llm embed-multi* - run bulk embeddings for multiple strings, using input from a CSV, TSV or JSON file, data from a SQLite database or data found by scanning the filesystem. [#215](#)
- *llm similar* - run similarity searches against your stored embeddings - starting with a search phrase or finding content related to a previously stored vector. [#190](#)
- *llm embed-models* - list available embedding models.
- `llm embed-db` - commands for inspecting and working with the default embeddings SQLite database.

There's also a new *llm.Collection* class for creating and searching collections of embedding from Python code, and a *llm.get_embedding_model()* interface for embedding strings directly. [#191](#)

2.13.9 0.8.1 (2023-08-31)

- Fixed bug where first prompt would show an error if the `io.datasette.llm` directory had not yet been created. [#193](#)
- Updated documentation to recommend a different `llm-gpt4all` model since the one we were using is no longer available. [#195](#)

2.13.10 0.8 (2023-08-20)

- The output format for `llm logs` has changed. Previously it was JSON - it's now a much more readable Mark-down format suitable for pasting into other documents. #160
 - The new `llm logs --json` option can be used to get the old JSON format.
 - Pass `llm logs --conversation ID` or `--cid ID` to see the full logs for a specific conversation.
- You can now combine piped input and a prompt in a single command: `cat script.py | llm 'explain this code'`. This works even for models that do not support *system prompts*. #153
- Additional *OpenAI-compatible models* can now be configured with custom HTTP headers. This enables platforms such as openrouter.ai to be used with LLM, which can provide Claude access even without an Anthropic API key.
- Keys set in `keys.json` are now used in preference to environment variables. #158
- The documentation now includes a *plugin directory* listing all available plugins for LLM. #173
- New *related tools* section in the documentation describing `ttok`, `strip-tags` and `symbex`. #111
- The `llm models`, `llm aliases` and `llm templates` commands now default to running the same command as `llm models list` and `llm aliases list` and `llm templates list`. #167
- New `llm keys` (aka `llm keys list`) command for listing the names of all configured keys. #174
- Two new Python API functions, `llm.set_alias(alias, model_id)` and `llm.remove_alias(alias)` can be used to configure aliases from within Python code. #154
- LLM is now compatible with both Pydantic 1 and Pydantic 2. This means you can install `llm` as a Python dependency in a project that depends on Pydantic 1 without running into dependency conflicts. Thanks, [Chris Mungall](#). #147
- `llm.get_model(model_id)` is now documented as raising `llm.UnknownModelError` if the requested model does not exist. #155

2.13.11 0.7.1 (2023-08-19)

- Fixed a bug where some users would see an `AlterError: No such column: log.id` error when attempting to use this tool, after upgrading to the latest `sqlite-utils 3.35` release. #162

2.13.12 0.7 (2023-08-12)

The new *Model aliases* commands can be used to configure additional aliases for models, for example:

```
llm aliases set turbo gpt-3.5-turbo-16k
```

Now you can run the 16,000 token `gpt-3.5-turbo-16k` model like this:

```
llm -m turbo 'An epic Greek-style saga about a cheesecake that builds a SQL database,
↳from scratch'
```

Use `llm aliases list` to see a list of aliases and `llm aliases remove turbo` to remove one again. #151

Notable new plugins

- **llm-mlc** can run local models released by the [MLC project](#), including models that can take advantage of the GPU on Apple Silicon M1/M2 devices.
- **llm-llama-cpp** uses `llama.cpp` to run models published in the GGML format. See [Run Llama 2 on your own Mac using LLM and Homebrew](#) for more details.

Also in this release

- OpenAI models now have min and max validation on their floating point options. Thanks, Pavel Král. #115
- Fix for bug where `llm templates list` raised an error if a template had an empty prompt. Thanks, Sherwin Daganato. #132
- Fixed bug in `llm install --editable` option which prevented installation of `.[test]`. #136
- `llm install --no-cache-dir` and `--force-reinstall` options. #146

2.13.13 0.6.1 (2023-07-24)

- LLM can now be installed directly from Homebrew core: `brew install llm`. #124
- Python API documentation now covers *System prompts*.
- Fixed incorrect example in the *Prompt templates* documentation. Thanks, Jorge Cabello. #125

2.13.14 0.6 (2023-07-18)

- Models hosted on [Replicate](#) can now be accessed using the `llm-replicate` plugin, including the new Llama 2 model from Meta AI. More details here: [Accessing Llama 2 from the command-line with the llm-replicate plugin](#).
- Model providers that expose an API that is compatible with the OpenAPI API format, including self-hosted model servers such as [LocalAI](#), can now be accessed using *additional configuration* for the default OpenAI plugin. #106
- OpenAI models that are not yet supported by LLM can also *be configured* using the new `extra-openai-models.yaml` configuration file. #107
- The `llm logs command` now accepts a `-m model_id` option to filter logs to a specific model. Aliases can be used here in addition to model IDs. #108
- Logs now have a SQLite full-text search index against their prompts and responses, and the `llm logs -q SEARCH` option can be used to return logs that match a search term. #109

2.13.15 0.5 (2023-07-12)

LLM now supports **additional language models**, thanks to a new *plugins mechanism* for installing additional models.

Plugins are available for 19 models in addition to the default OpenAI ones:

- `llm-gpt4all` adds support for 17 models that can download and run on your own device, including Vicuna, Falcon and wizardLM.
- `llm-mpt30b` adds support for the MPT-30B model, a 19GB download.
- `llm-palm` adds support for Google's PaLM 2 via the Google API.

A comprehensive tutorial, *writing a plugin to support a new model* describes how to add new models by building plugins in detail.

New features

- *Python API* documentation for using LLM models, including models from plugins, directly from Python. #75
- Messages are now logged to the database by default - no need to run the `llm init-db` command any more, which has been removed. Instead, you can toggle this behavior off using `llm logs off` or turn it on again using `llm logs on`. The `llm logs status` command shows the current status of the log database. If logging is turned off, passing `--log` to the `llm prompt` command will cause that prompt to be logged anyway. #98
- New database schema for logged messages, with `conversations` and `responses` tables. If you have previously used the old `logs` table it will continue to exist but will no longer be written to. #91
- New `-o/--option name value` syntax for setting options for models, such as temperature. Available options differ for different models. #63
- `llm models list --options` command for viewing all available model options. #82
- `llm "prompt" --save template` option for saving a prompt directly to a template. #55
- Prompt templates can now specify *default values* for parameters. Thanks, Chris Mungall. #57
- `llm openai models` command to list all available OpenAI models from their API. #70
- `llm models default MODEL_ID` to set a different model as the default to be used when `llm` is run without the `-m/--model` option. #31

Smaller improvements

- `llm -s` is now a shortcut for `llm --system`. #69
- `llm -m 4-32k` alias for `gpt-4-32k`.
- `llm install -e directory` command for installing a plugin from a local directory.
- The `LLM_USER_PATH` environment variable now controls the location of the directory in which LLM stores its data. This replaces the old `LLM_KEYS_PATH` and `LLM_LOG_PATH` and `LLM_TEMPLATES_PATH` variables. #76
- Documentation covering *Utility functions for plugins*.
- Documentation site now uses Plausible for analytics. #79

2.13.16 0.4.1 (2023-06-17)

- LLM can now be installed using Homebrew: `brew install simonw/llm/llm`. #50
- `llm` is now styled LLM in the documentation. #45
- Examples in documentation now include a copy button. #43
- `llm templates` command no longer has its display disrupted by newlines. #42
- `llm templates` command now includes system prompt, if set. #44

2.13.17 0.4 (2023-06-17)

This release includes some backwards-incompatible changes:

- The `-4` option for GPT-4 is now `-m 4`.
- The `--code` option has been removed.
- The `-s` option has been removed as streaming is now the default. Use `--no-stream` to opt out of streaming.

Prompt templates

Prompt templates is a new feature that allows prompts to be saved as templates and re-used with different variables.

Templates can be created using the `llm templates edit` command:

```
llm templates edit summarize
```

Templates are YAML - the following template defines summarization using a system prompt:

```
system: Summarize this text
```

The template can then be executed like this:

```
cat myfile.txt | llm -t summarize
```

Templates can include both system prompts, regular prompts and indicate the model they should use. They can reference variables such as `$input` for content piped to the tool, or other variables that are passed using the new `-p/--param` option.

This example adds a voice parameter:

```
system: Summarize this text in the voice of $voice
```

Then to run it (via `strip-tags` to remove HTML tags from the input):

```
curl -s 'https://til.simonwillison.net/macos/imovie-slides-and-audio' | \
strip-tags -m | llm -t summarize -p voice GladOS
```

Example output:

```
My previous test subject seemed to have learned something new about iMovie. They exported keynote slides as individual images [...] Quite impressive for a human.
```

The *Prompt templates* documentation provides more detailed examples.

Continue previous chat

You can now use `llm` to continue a previous conversation with the OpenAI chat models (`gpt-3.5-turbo` and `gpt-4`). This will include your previous prompts and responses in the prompt sent to the API, allowing the model to continue within the same context.

Use the new `-c/--continue` option to continue from the previous message thread:

```
llm "Pretend to be a witty gerbil, say hi briefly"
```

```
Greetings, dear human! I am a clever gerbil, ready to entertain you with my quick wit and endless energy.
```

```
llm "What do you think of snacks?" -c
```

Oh, how I adore snacks, dear human! Crunchy carrot sticks, sweet apple slices, and chewy yogurt drops are some of my favorite treats. I could nibble on them all day long!

The `-c` option will continue from the most recent logged message.

To continue a different chat, pass an integer ID to the `--chat` option. This should be the ID of a previously logged message. You can find these IDs using the `llm logs` command.

Thanks [Amjith Ramanujam](#) for contributing to this feature. #6

New mechanism for storing API keys

API keys for language models such as those by OpenAI can now be saved using the new `llm keys` family of commands.

To set the default key to be used for the OpenAI APIs, run this:

```
llm keys set openai
```

Then paste in your API key.

Keys can also be passed using the new `--key` command line option - this can be a full key or the alias of a key that has been previously stored.

See [link-to-docs](#) for more. #13

New location for the logs.db database

The `logs.db` database that stores a history of executed prompts no longer lives at `~/ .llm/log.db` - it can now be found in a location that better fits the host operating system, which can be seen using:

```
llm logs path
```

On macOS this is `~/Library/Application Support/io.datasette.llm/logs.db`.

To open that database using Datasette, run this:

```
datasette "$(llm logs path)"
```

You can upgrade your existing installation by copying your database to the new location like this:

```
cp ~/.llm/log.db "$(llm logs path)"  
rm -rf ~/.llm # To tidy up the now obsolete directory
```

The database schema has changed, and will be updated automatically the first time you run the command.

That schema is [included in the documentation](#). #35

Other changes

- New `llm logs --truncate` option (shortcut `-t`) which truncates the displayed prompts to make the log output easier to read. #16
- Documentation now spans multiple pages and lives at <https://llm.datasette.io/> #21
- Default `llm chatgpt` command has been renamed to `llm prompt`. #17
- Removed `--code` option in favour of new prompt templates mechanism. #24
- Responses are now streamed by default, if the model supports streaming. The `-s/--stream` option has been removed. A new `--no-stream` option can be used to opt-out of streaming. #25
- The `-4/--gpt4` option has been removed in favour of `-m 4` or `-m gpt4`, using a new mechanism that allows models to have additional short names.
- The new `gpt-3.5-turbo-16k` model with a 16,000 token context length can now also be accessed using `-m chatgpt-16k` or `-m 3.5-16k`. Thanks, Benjamin Kirkbride. #37
- Improved display of error messages from OpenAI. #15

2.13.18 0.3 (2023-05-17)

- `llm logs` command for browsing logs of previously executed completions. #3
- `llm "Python code to output factorial 10" --code` option which sets a system prompt designed to encourage code to be output without any additional explanatory text. #5
- Tool can now accept a prompt piped directly to standard input. #11

2.13.19 0.2 (2023-04-01)

- If a SQLite database exists in `~/.llm/log.db` all prompts and responses are logged to that file. The `llm init-db` command can be used to create this file. #2

2.13.20 0.1 (2023-04-01)

- Initial prototype release. #1