Member-only story

Superfast LLM text generation with vLLM on Windows 11

Learn to install and use vLLM with OpenAI, LangChain, and Guidance AI on a Windows 11 PC.

Jes Fink-Jensen · Follow

Published in Better Programming

13 min read · Sep 15, 2023

Photo by Aleksei Sabulevskii on Unsplash

In this article, I will show you how to install vLLM on a Windows 11 PC, so that you can run your local Large Language Models (LLMs) faster than with other solutions, like Oobabooga, for example.

Once vLLM is installed you'll be able to serve your local LLMs like MosaicML's MPT or Meta's Llama 2 easily. I will show you how to connect to a vLLM server using three different Python libraries for working with LLMs: OpenAI, LangChain, and Guidance AI.

This writeup is a result of my own experimentation with vLLM and the Python libraries that I mentioned. I would also like to note that my GPU is an NVidia Quadro RTX a4500 with 20GB of memory. The GPU memory is thus large enough to run the smallest versions of MPT, Llama 2, and the like. So, please make sure that the memory of your GPU is also large enough to contain these LLM models.

Installing vLLM on a Windows 11 PC

First, I tried to install vLLM using the standard Windows command line interpreter. This didn't work.

Therefore, I decided to try to install vLLM in the Windows Linux subsystem (WSL — version 2) that was running Ubuntu 22.04. This worked fine. Below, I show how I managed to do this.

First, I logged in to my freshly installed WSL account from the Windows command line interface (cmd):

```
wsl -u <username>
```

Next, I installed CUDA 11.8 for Ubuntu-WSL as explained on the NVidia download page:

```
wget https://developer.download.nvidia.com/compute/cuda/repos/wsl-ubuntu/x86_64
sudo mv cuda-wsl-ubuntu.pin /etc/apt/preferences.d/cuda-repository-pin-600
wget https://developer.download.nvidia.com/compute/cuda/11.8.0/local_installers
sudo dpkg -i cuda-repo-wsl-ubuntu-11-8-local_11.8.0-1_amd64.deb
sudo cp /var/cuda-repo-wsl-ubuntu-11-8-local/cuda-*-keyring.gpg /usr/share/keyr
```

```
sudo apt-get update
sudo apt-get -y install cuda
```

Apparently, PIP was not installed on Ubuntu, so I did that as well:

```
sudo apt install python3-pip
```

Then, I finally was able to install the vLLM Python library:

```
pip3 install vllm
```

The first time I tried to run vLLM in WSL, there were some dependencies missing. Hence, while still in WSL, I cloned the Git repo containing vLLM and installed all the required libraries:

```
cd ~
git clone https://github.com/vllm-project/vllm.git
cd vllm
sudo pip3 install -e .
```

Serving an LLM with vLLM in WSL

When everything has been installed, we are ready to run a vLLM server. So, to run a vLLM server in WSL, we can now write the following:

```
python3 -m vllm.entrypoints.openai.api_server
```

This starts a vLLM server that uses part of the OpenAI API. The great thing about this is that code that was originally made to run with OpenAI GPT models, can also be made to work with the vLLM model that we are currently serving.

Also, when running the server for the first time, it will download an LLM model and tokenizer called `facebook/opt-125m` by default. If you do not wish to use this model, simply do `ctrl+C` to quit.

Given that I had previously installed Oobabooga, I had already downloaded some models that I wanted to use. These models are located in the following location on my PC: `D:\ooba\text-generation-webui\models`.

The models that I have used with vLLM are the following:

- **MosaicML MPT-7B Chat** (in subdirectory: `mosaicml_mpt-7b-chat`)
- **MosaicML MPT-7B Instruct** (in subdirectory: `mosaicml_mpt-7b-instruct`)
- **Llama 2 — 7B — HF** (in subdirectory: `Llama-2-7b-hf`)
- **Llama 2–7B — Chat—HF** (in subdirectory: `Llama-2-7b-chat-hf`)

These models can all be downloaded from Huggingface.

So, to run the vLLM server with, for example, the MosaicML MPT-7B Chat LLM model, I write the following:

```
python3 -m vllm.entrypoints.openai.api_server --model /mnt/d/ooba/text-generati
```

Here are three things to note:

- I have translated the Windows directory path `D:\ooba\text-generation-webui\models\mosaicml_mpt-7b-chat` into a Linux style path: `/mnt/d/ooba/text-generation-webui/models/mosaicml_mpt-7b-chat`. In your case, you'll need to take note of where your downloaded models reside on your Windows PC and then also translate the path to Linux style.
- I have added the `--trust-remote-code` flag. This is necessary when running models like the ones that I have mentioned.
- Finally, I have added `--port 9999` to make sure that the server is running on a port that is not typically used by other software.

The result is the following output:

```
INFO 09-14 18:28:51 llm_engine.py:60] Initializing an LLM engine with config: model='/mnt/d/ooba/text-generation-webui/models/mosaicml_mpt-7b-chat', tokenizer='/mnt/d/ooba/text-generation-webui/models/mosaicml_mpt-7b-chat', tokenizer_mode=auto, trust_remote_code=True, dtype=torch.bfloat16, use_dummy_weights=False, download_dir=None, use_np_weights=False, tensor_parallel_size=1, seed=0)
INFO 09-14 18:30:05 llm_engine.py:134] # GPU blocks: 608, # CPU blocks: 512
WARNING 09-14 18:30:06 cache_engine.py:96] Using 'pin_memory=False' as WSL is detected. This may slow down the performance.
INFO:     Started server process [963]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://localhost:9999 (Press CTRL+C to quit)
```

Running vLLM with MosaicML MPT 7B Chat on WSL

Next, we can try to connect to this server. If this doesn't work straight away, you may want to check out the troubleshooting part later in this article.

Running a model with vLLM and OpenAI

The most straightforward way to connect to the vLLM server is by using the OpenAI library. This is also what is suggested in the documentation of vLLM.

To install the OpenAI libraries, you can simply write:

```
pip install openai
```

The OpenAI github page lists other possibilities, so you may want to check those out as well.

Next are two pieces of code showing how to connect to the vLLM server.

A code example using OpenAI

```python
import openai

base_url = "http://localhost:9999/v1"
openai.api_key = "***"
openai.api_base = base_url

models = openai.Model.list()
model = models["data"][0]["id"]
print(f"model: {model}")

prompt = "What is the capital of France?"
stream = False

completion = openai.Completion.create(
    model=model,
    prompt=prompt,
    echo=False,
    max_tokens=1500,
    temperature=0.0,
    frequency_penalty=1.1,
    stream=stream)

print(completion["choices"][0]["text"])
```

vllm_openai_api.py hosted with ❤️ by GitHub view raw

A description of the OpenAI code example

Here's a step-by-step description of the above code:

Importing the Library:

- The `openai` library is imported.

Setting the Base URL and API Key:

- A variable named `base_url` is initialized with the value `http://localhost:9999/v1`. This is the URL of our vLLM server running an LLM using the OpenAI API.
- The API key for the OpenAI library `openai.api_key` is set to a placeholder value `***`, as we don't need it.
- The base URL `openai.api_base` for the OpenAI API is set to the previously defined `base_url`.

Fetching the Model Name:

- The `openai.Model.list()` method is called to retrieve a list of available models from the API. The results are stored in the `models` variable.
- From the `models` list, the name of the first (and only) model is extracted and stored in the `model` variable.
- The `model` is then printed to the console so we can verify it.

Setting the Prompt and Stream Variable:

- A variable named `prompt` is initialized with the question string, "What is the capital of France?".
- A variable named `stream` is set to `False`. This indicates that the response from the API should not be streamed.

Generating a Completion:

- The `openai.Completion.create()` method is called to generate a completion (or response) based on the provided prompt.
- Several parameters are passed to this method:
  - `model`: The model name extracted earlier.
  - `prompt`: The question string about the capital of France.
  - `echo`: Set to `False`, indicating that the input prompt should not be echoed in the output.
  - `max_tokens`: The maximum number of tokens in the response is set to 1500.
  - `temperature`: Set to 0.0, which means the output will be deterministic and less random.
  - `frequency_penalty`: Set to 1.1, which will penalize more frequent tokens in the output.
  - `stream`: Whether the response should be streamed or not (set to `False`).

Printing the Completion:

- The generated completion (or answer) is extracted from the `completion` dictionary using the keys `["choices"][0]["text"]`.
- The answer is then printed to the console.

The output of the OpenAI example

In the screenshot below, you can see the name of the model as well as the output of the completion:

```
model: /mnt/d/ooba/text-generation-webui/models/mosaicml_mpt-7b-chat

What is the capital of France?
The capital of France is Paris.
```

A Chat code example

It is also possible to use the OpenAI functions specifically for use with chat-oriented LLMs.

```python
import openai

base_url = "http://localhost:9999/v1"
openai.api_key = "***"
openai.api_base = base_url

models = openai.Model.list()
model = models["data"][0]["id"]
print(f"model: {model}")

prompt = "What is the capital of France?"
messages = [{"role": "user", "content": prompt}]

chat_completion = openai.ChatCompletion.create(
    model=model,
    messages=messages,
    echo=False,
    max_tokens=1500,
    temperature=0.0,
    frequency_penalty=1.1)

question = messages[0]
answer = chat_completion["choices"][0]["message"]
print(f"{question['role']}: {question['content']}")
print(f"{answer['role']}: {answer['content']}")
```

vllm_openai_chat_api.py hosted with ❤ by GitHub view raw

A comparison of the standard code and the chat code

The OpenAI chat code is essentially the same as the standard OpenAI code except for a few differences. Let's compare the first and second pieces of code to identify the main differences:

Completion vs. ChatCompletion:

- In the first code, the method `openai.Completion.create()` is used. This method is designed for generating completions or responses based on a single prompt.
- In the second code, the method `openai.ChatCompletion.create()` is used. This method is designed for a chat-based interaction where a series of messages can be provided, and the model responds in a conversational manner.

Prompt Structure:

- In the first code, the prompt is a simple string: `prompt = "What is the capital of France?"`.
- In the second code, the prompt is structured as a list of messages: `messages = [{"role": "user", "content": prompt}]`. This format allows for a more interactive and conversational approach, where multiple messages can be added to simulate a back-and-forth conversation.

Printing the Output:

- In the first code, the output is directly printed using: `print(completion["choices"][0]["text"])`.
- In the second code, both the question and the answer are printed in a chat format. The roles (“user” and “assistant”) and their respective content are extracted and printed in a conversational manner.

Output Extraction:

- In the first code, the output is extracted using `completion["choices"][0]["text"]`.
- In the second code, the output is extracted using `chat_completion["choices"][0]["message"]`, which provides more details about the message, including the role ("user" or "assistant") and the content of the message.

Here, you can see that the second code is designed for a more complex interaction, while the first code is simpler in its approach.

The output of the OpenAI Chat example

```
model: /mnt/d/ooba/text-generation-webui/models/mosaicml_mpt-7b-chat
user: What is the capital of France?
assistant: The capital of France is Paris.
```

Running a model with vLLM and LangChain

One of the most popular frameworks for working with LLMs is LangChain. It is perfectly possible to connect to a vLLM server using the functionality provided by LangChain.

It can be installed by simply writing:

```
pip install langchain
```

Below are two examples of how you can connect to vLLM using either a standard method LangChain/OpenAI or a more dedicated LangChain/ChatOpenAI method.

A LangChain/OpenAI code example

```python
from langchain import PromptTemplate
from langchain.chains import LLMChain
from langchain.llms import OpenAI

import requests

base_url = "http://localhost:9999/v1"
x = requests.get(base_url + "/models")
model = str(x.json()["data"][0]["id"])
print(f"model: {model}")

llm = OpenAI(temperature=0.7,
         frequency_penalty=1.1,
         openai_api_key="***",
         verbose=True,
         openai_api_base = base_url,
         model_name = model,
         max_tokens = 1024,
         logit_bias = None)

prompt = PromptTemplate(
    input_variables=["country"],
    template="What is the capital of {country}?"
)

chain = LLMChain(llm=llm, prompt=prompt)

print(chain.run("France"))
```

langchain_openai.py hosted with ❤️ by GitHub view raw

A description of the code example

Here's a description of the code from above:

Importing Libraries and Modules:

- Modules from the `langchain` library are imported: `PromptTemplate`, `LLMChain`, and `OpenAI`.
- The `requests` library is imported, which is commonly used for making HTTP requests in Python.

Setting the Base URL and Fetching Model Name:

- The `base_url` variable is initialized with the value `http://localhost:9999/v1`, which refers to our local vLLM server OpenAI API endpoint.

- An HTTP GET request is made to the endpoint `base_url + "/models"` to fetch a list of available LLM models.
- The name of the first model from the response is extracted and stored in the `model` variable.
- The model name is then printed to the console.

Initializing the OpenAI LLM (Language Model):

- An instance of the `OpenAI` class from the `langchain` library is created and stored in the `llm` variable.
- Several parameters are passed to configure the language model:
  - `temperature` : Determines the randomness of the output.
  - `frequency_penalty` : Penalizes more frequent tokens in the output.
  - `openai_api_key` : A placeholder API key.
  - `verbose` : Set to `True` , enabling detailed output.
  - `openai_api_base` : The base URL `base_url` for the OpenAI API.
  - `model_name` : The model name `model` fetched earlier.
  - `max_tokens` : The maximum number of tokens in the response.
  - `logit_bias` : Set to `None` .

Setting up the Prompt Template:

- An instance of the `PromptTemplate` class is created with the name `prompt` .
- The template is designed to ask about the capital of a given country. The country is an input variable, represented by `{country}` in the template string.

Creating the LLM Chain:

- An instance of the `LLMChain` class is created with the name `chain` .
- The previously initialized language model (`llm`) and prompt template (`prompt`) are passed as parameters. This chain will use the specified language model and prompt to generate responses.

Running the Chain and Printing the Result:

- The `chain.run()` method is called with the argument "France" which means the chain will generate a response to the question about the capital of France.
- The result of this execution is printed to the console.

The output of the LangChain/OpenAI example

First, the name of the current model being served on the vLLM server is printed, which is then followed by the output of the LLM Chain.

```
model: /mnt/d/ooba/text-generation-webui/models/mosaicml_mpt-7b-chat

What is the capital of France?
The capital of France is Paris.
Paris is a city located in the north-central part of France, on the Seine river. It is one of Europe's most visited citi
es, known for its rich history, cultural landmarks and artistic heritage. The city has been ruled by many different peop
le over the centuries, making it a melting pot of cultures and architecture. Paris was founded as a settlement around 35
0 BC by Celtic Gauls and became an important center for trade during Roman rule. The city has since been through many hi
storical periods including the Middle Ages and Renaissance, becoming a major center for art and architecture during this
 time period. Today, Paris remains an important cultural hub with many famous landmarks such as the Eiffel Tower, Notre-
Dame Cathedral and Louvre Museum drawing millions of visitors each year from around the world.
French culture is also renowned worldwide for its cuisine (including baguettes, croissants, escargot), fashion (Chanel N
o 5) and philosophy (René Descartes). Other notable aspects include French language (often considered one of Europe's mo
st beautiful languages), music (from composers like Claude Debussy to singers like Edith Piaf) ... all these elements ar
e deeply rooted in French history that dates back to ancient Gaul!
```

The response from the chain not only answers the question in the prompt, but also provides a lot of additional information that we did not ask for. This seems to be a drawback of using this particular method.

A LangChain/ChatOpenAI code example

```python
from langchain import PromptTemplate
from langchain.prompts.chat import ChatPromptTemplate, HumanMessagePromptTemplate
from langchain.chains import LLMChain
from langchain.chat_models import ChatOpenAI

import requests

base_url = "http://localhost:9999/v1"
x = requests.get(base_url + "/models")
model = str(x.json()["data"][0]["id"])
print(f"model: {model}")

llm = ChatOpenAI(temperature=0.7,
         openai_api_key="***",
         verbose=True,
         openai_api_base = base_url,
         model_name = model,
         max_tokens = 1024)

prompt = PromptTemplate(
    input_variables=["country"],
    template="What is the capital of {country}?"
)

message_prompt = HumanMessagePromptTemplate(prompt=prompt)
chat_prompt = ChatPromptTemplate.from_messages([message_prompt])

chain = LLMChain(llm=llm, prompt=chat_prompt)

print(chain.run("France"))
```

langchain_chatopenai.py hosted with ❤ by GitHub view raw

A comparison of the code examples

Let's compare the two pieces of code to identify the main differences:

Import Statements:

- In the second code, two additional modules are imported: from `langchain.prompts.chat` the modules `ChatPromptTemplate` and `HumanMessagePromptTemplate` are imported.
- Instead of importing `OpenAI` from `langchain.llms` as in the first code, the second code imports `ChatOpenAI` from `langchain.chat_models`.

Language Model Initialization:

- In the first code, the language model is initialized using the `OpenAI` class.
- In the second code, the language model is initialized using the `ChatOpenAI` class, which is designed for chat-based interactions.

Prompt Setup:

- In the first code, a simple `PromptTemplate` is used to create the prompt.
- In the second code, while the base `PromptTemplate` is still used, it is further wrapped inside a `HumanMessagePromptTemplate`. This human message prompt is then used to create a `ChatPromptTemplate` using the `from_messages` method. This setup is more complex and is designed to simulate a chat-based interaction.

Chain Initialization:

- In both codes, an `LLMChain` is initialized. However, the prompt passed to the chain in the second code is the chat-based `chat_prompt`, while in the first code, it's the simpler `prompt`.

So, while both codes interact with the OpenAI API on the vLLM server to get a response to the question about the capital of France, the main differences lie in the setup and structure of the prompts and the type of language model used.

The second code is designed for a more chat-based interaction, simulating a conversation, while the first code uses a more straightforward prompt-based approach.

This is analogous to the code using the 'pure' OpenAI library.

The output of the LangChain/ChatOpenAI code

Below, we can see in the screenshot that the correct model is run and that the LLM chain is returning the correct answer, without any additional unwanted information as was the case in the other LangChain code example.

```
model: /mnt/d/ooba/text-generation-webui/models/mosaicml_mpt-7b-chat
The capital of France is Paris.
```

Running a model with vLLM and Guidance AI

It is also possible to access the vLLM server using Guidance AI.

Installing Guidance AI

If the latest release of Guidance AI is 0.0.64 as you read this, then write the following to install the version that allows the use of non-OpenAI LLMs:

```
pip install -U git+https://github.com/guidance-ai/guidance@23d0ba12720d09bb87b5
```

Otherwise, if the release is later than 0.0.64, then you may just write:

```
pip install guidance
```

A code example

```python
import guidance
import requests

base_url = "http://localhost:9999/v1"
x = requests.get(base_url + "/models")
model = str(x.json()["data"][0]["id"])

guidance.llm = guidance.llms.OpenAI(
        model=model,
        endpoint=base_url,
        api_key="***",
        chat_mode=False,
        encoding_name="cl100k_base"
)

program = guidance(
        """What are the top five fun facts about {{country}}? Provide a one-liner with a descri

    Here are the fun facts:
    {{~#geneach 'facts' num_iterations=5}}
    [{{@index}}]: "{{gen 'this' stop='"' max_tokens=50 ~}}"
    {{/geneach}}
        """
)

result = program(country="France")
print(result["facts"])
print("==========================================")
print(result)
```

vllm_guidance.py hosted with ❤ by GitHub view raw

Description of the Guidance AI example

Here's a step-by-step description of what the above code is doing:

Importing Libraries:

- The `guidance` library is imported. This Python library contains all the functionality of Guidance AI.
- The `requests` library is imported, which is a popular Python library for making HTTP requests.

Setting the Base URL:

- A variable named `base_url` is initialized with the value "http://localhost:9999/v1". This URL refers to our vLLM server API endpoint.

Fetching Model Name:

- An HTTP GET request is made to the endpoint `base_url + "/models"`, which translates to "http://localhost:9999/v1/models".
- The response from this request is stored in the variable `x`.
- The JSON response from `x` is parsed to extract the name of the LLM model that is being served by the vLLM server. This model name is stored in the variable `model`.

Setting up the Guidance Model:

- The `guidance.llm` is initialized using the `guidance.llms.OpenAI` class.
- Several parameters are passed to this class:
  - `model`: The model name fetched in the previous step.
  - `endpoint`: The base URL `base_url` for the API.
  - `api_key`: A placeholder API key (represented as "***"), as we do not need an OpenAI key for running our own local LLM models.
  - `chat_mode`: Set to `False`, as this seems to give the best results — even when using a Chat LLM model.
  - The `encoding_name` is set to `cl100k_base` to let Guidance AI know which encoding is to be used for the internal tokenizer.

Defining the Program:

- A variable named `program` is initialized with a string template that is a prompt for generating content with Guidance AI.
- The template asks for the top five fun facts about a country (specified by the placeholder `{{country}}`).
- The template uses a loop (`{{~#geneach 'facts' num_iterations=5}}`) to generate five fun facts. Each fact is generated using the `gen` function and is limited to 50 tokens.

Executing the Program:

- The `program` is executed with the argument `country="France"`, which means it will generate fun facts about France.
- The result of this execution is stored in the variable `result`.

Printing the Results:

- The fun facts about France (stored in `result["facts"]`) are printed to the console.
- A separator line ("==========") is printed.
- The entire `result` dictionary is printed to the console, showing all the data it contains.

The output of the Guidance AI example

Below is a screenshot, showing the output when running the example:

```
["The Eiffel Tower was built for the 1889 World's Fair and is one of the most recognizable landmarks in the world.", 'The French Revolution began
in 1789 and led to the end of the monarchy and the rise of a republic.', 'The French language is one of the most widely spoken languages in the wo
rld.', 'The Louvre Museum in Paris is one of the most visited art museums in the world.', 'The French cuisine is known for its rich and flavorful
dishes, including croissants, escargot, and wine.']
========================================
What are the top five fun facts about France? Provide a one-liner with a description of each of the facts.

    Here are the fun facts:
    [0]: "The Eiffel Tower was built for the 1889 World's Fair and is one of the most recognizable landmarks in the world."

    [1]: "The French Revolution began in 1789 and led to the end of the monarchy and the rise of a republic."

    [2]: "The French language is one of the most widely spoken languages in the world."

    [3]: "The Louvre Museum in Paris is one of the most visited art museums in the world."

    [4]: "The French cuisine is known for its rich and flavorful dishes, including croissants, escargot, and wine."
```

Troubleshooting the connection to the vLLM server

It may happen to you, as it did to me at first, that you may not be able to connect to the vLLM server from the normal Windows environment.

First, to test if the vLLM server is running correctly in WSL, we can open another WSL terminal and try to connect from there. We can run the following as a test:

```
curl http://localhost:9999/v1/models
```

If this does not give an error, then we know that the server is running as it should.

So why can we not connect from the Windows environment?

It is likely due to a port-forwarding problem in WSL:

'Seems like a forwarding problem. WSL2's interface is NAT'd, whereas WSL1 was bridged by default. WSL seems to do some "auto-forwarding" of ports, but only on localhost. However, sometimes this auto-forwarding mechanism seems to "break down". The main culprit seems to be hibernation or Windows Fast Startup (which are both closely-related features).'

Luckily, the solution is simple. Write the following on the Windows command line (cmd) and restart either the WSL session or your PC (whatever works):

```
wsl --shutdown
```

Now, after each WSL session, you'll have to write the above statement to make sure that the ports are correctly forwarded the next time that you start a WSL session.

References

vLLM documentation

OpenAI API reference

LangChain documentation

Guidance AI documentation

"Install specific git commit with pip" on StackOverflow

"Get err_connection_refused accessing django running on wsl2 from Windows but can curl from Windows terminal" on StackOverflow

Large Language Models | Text Generation | Python | OpenAI | Langchain

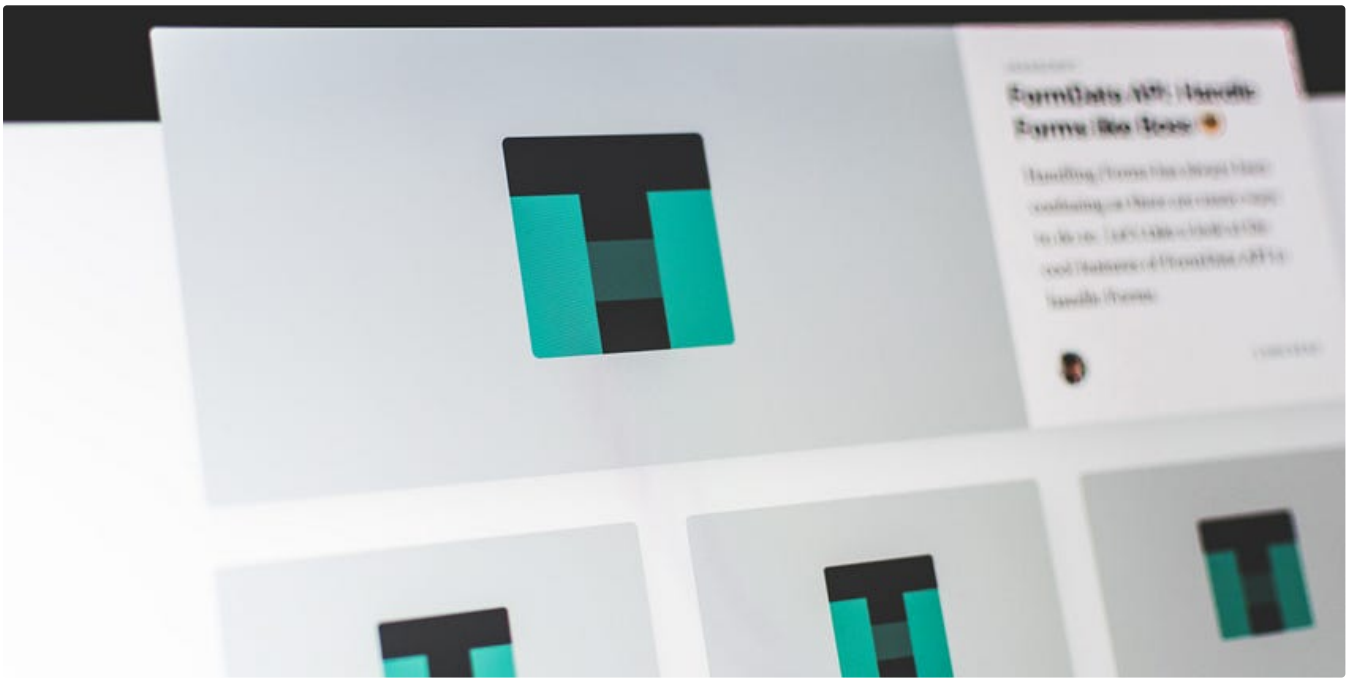
Follow

Written by Jes Fink-Jensen

352 Followers · Writer for Better Programming

Not your average geek...

More from Jes Fink-Jensen and Better Programming



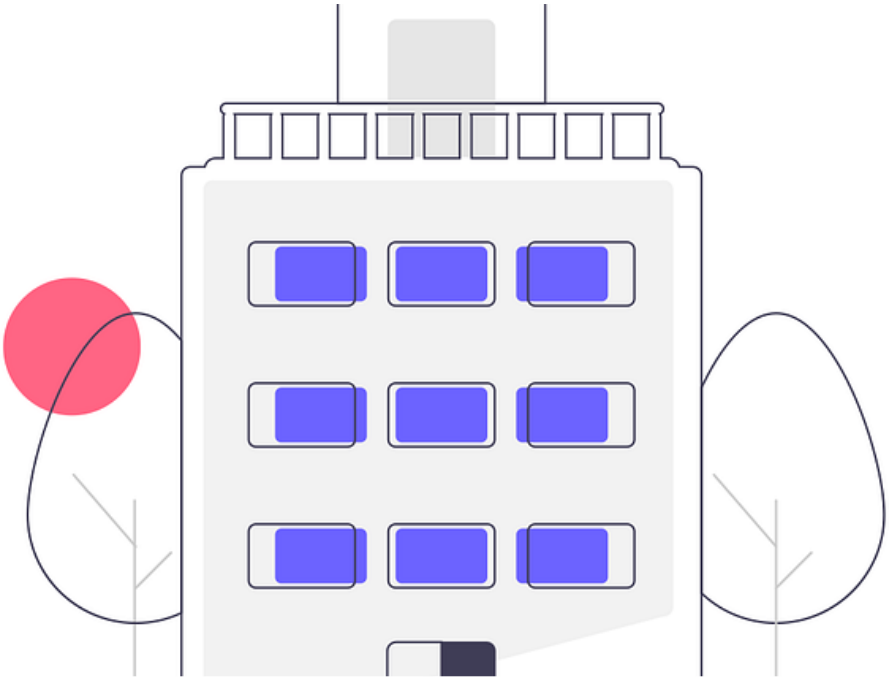
Jes Fink-Jensen in Better Programming

How to Generate HTML With Golang Templates

An example of how to use Go templates for generating an HTML page

· 6 min read · Mar 12, 2022

195



Bharath in Better Programming

The Clean Architecture — Beginner's Guide

As explained with visual illustrations

6 min read · Jan 4, 2022

2.9K 20

Peter Norvig in Better Programming

New Python Operators!

The new “walrus operator” in Python 3.8, written as :=, has been much discussed. This post introduces additional whimsically-named...

3 min read · Apr 4, 2023

912 9

Jes Fink-Jensen in Better Programming

How To Render HTML Pages With Gin For Golang

A simple example that shows how to render HTML template pages using the popular web framework Gin for the Go Language (Golang)

· 4 min read · Apr 22, 2022

140

See all from Jes Fink-Jensen

See all from Better Programming

Recommended from Medium

Feature	TGI - Text Generation Interface	vLLM - Versatile Large Language
llelism	✓ Utilizes Tensor Parallelism for faster inference	✓ Leverages Tensor Parallelism for high throug
Transformers Code	✓ Optimizes transformer code for inference	✓ Employs optimized CUDA kernels for speed
on	✓ Supports quantization methods	✗ Does not support quantization
Batching	✓ Batches incoming requests continuously	✓ Maximizes GPU utilization with continuous
l Weight Loading	✓ Accelerated weight loading for quick startup	✗ Does not include accelerated weight loadin
oing	✓ Offers logits warping options	✗ Does not provide logits warping features
mpt Generation	✓ Allows custom prompt generation	✗ Does not support custom prompt generatio
Support	✓ Supports fine-tuned models for higher accuracy	✗ Lacks fine-tuning support
ghput	Moderate throughput	Excellent serving throughput
ntion	✗ Does not include Paged Attention	✓ Efficiently manages memory with Paged Att
Outputs	✓ Supports streaming outputs	✓ Enables streaming outputs for improved pe
Various Models	✓ Compatible with various LLMs	✓ Seamlessly supports a wide range of Huggir

Rohit Kewalramani

TGI vs. vLLM: Making Informed Choices for LLM Deployment

In the world of deploying and serving Large Language Models (LLMs), two notable frameworks have emerged as powerful solutions: Text...

3 min read · Sep 24, 2023

 21  1

Eda Johnson in Snowflake

Generating Product Descriptions with Mistral-7B-Instruct-v0.2 with vLLM Serving

Date: December 2023

5 min read · Dec 20, 2023

Phillip Gimmi

What is GGUF and GGML?

GGUF and GGML are file formats used for storing models for inference, especially in the context of language models like GPT (Generative...

2 min read · Sep 8, 2023

368

BoredGeekSociety

Open Source LLM SQL Coder beats GPT-4? Everything You Need to Know!

It finally happened! An open sourced fine-tuned model becomes the best Coding LLM, beating GPT-4! Everything you need to know!

· 3 min read · Feb 5, 2024

72

Datadrifters in Dev Genius

vLLM + AutoAWQ: Fastest Way To Serve LLMs

Join our next cohort: Full-stack GenAI SaaS Product in 4 weeks!

· 7 min read · Nov 3, 2023

170 1

Parikshit Saikia

Mistral Mastery: Fine-Tuning & Fast Inference Guide

Unlocking full potential of Mistral-7B-Inst with QLoRA and vLLM

8 min read · Oct 30, 2023