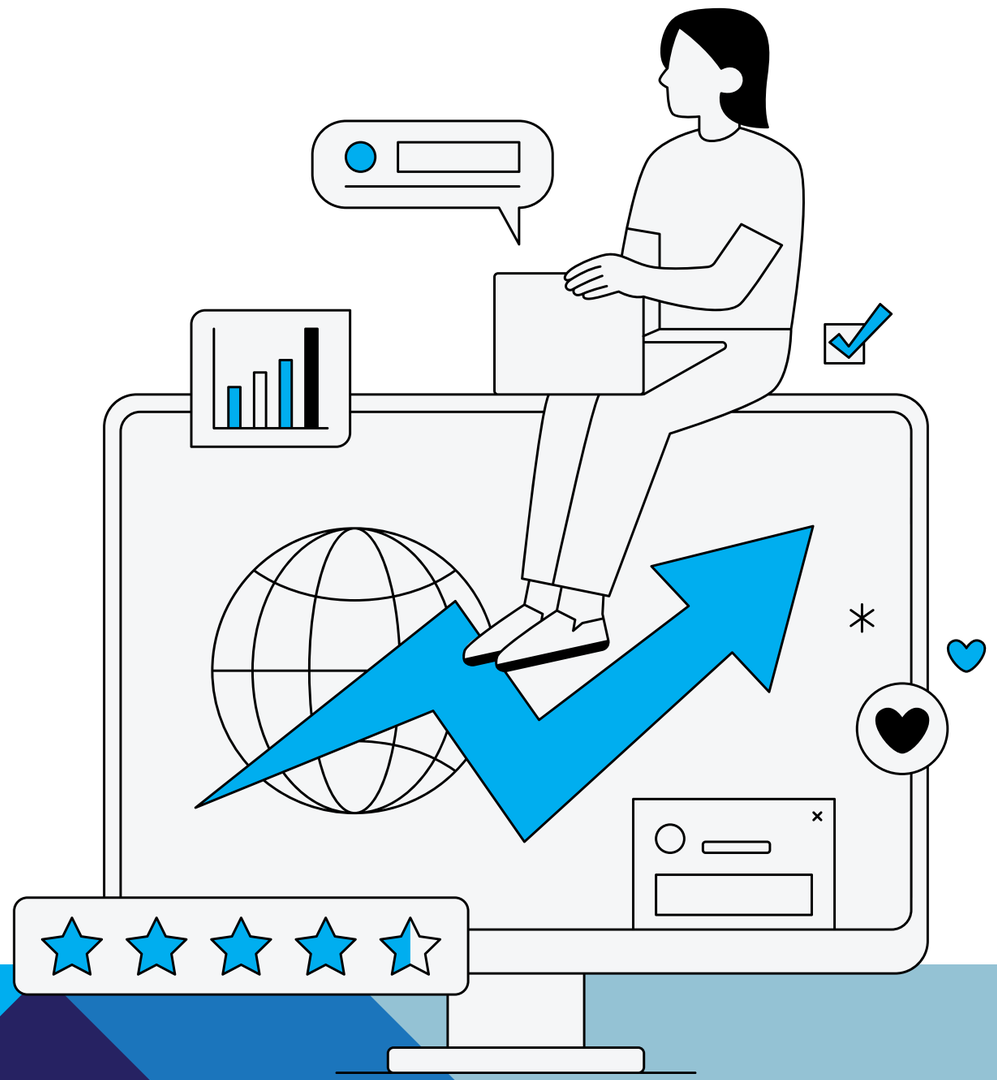
From Development to Production: Deploying Large Language Model (Llama3) Successfully



This document includes LLM deployment steps in detail:

1. Running Llama 3 model locally
2. Deploying the model using Ngrok proxy server

-Amrita Rath

LLM Deployment

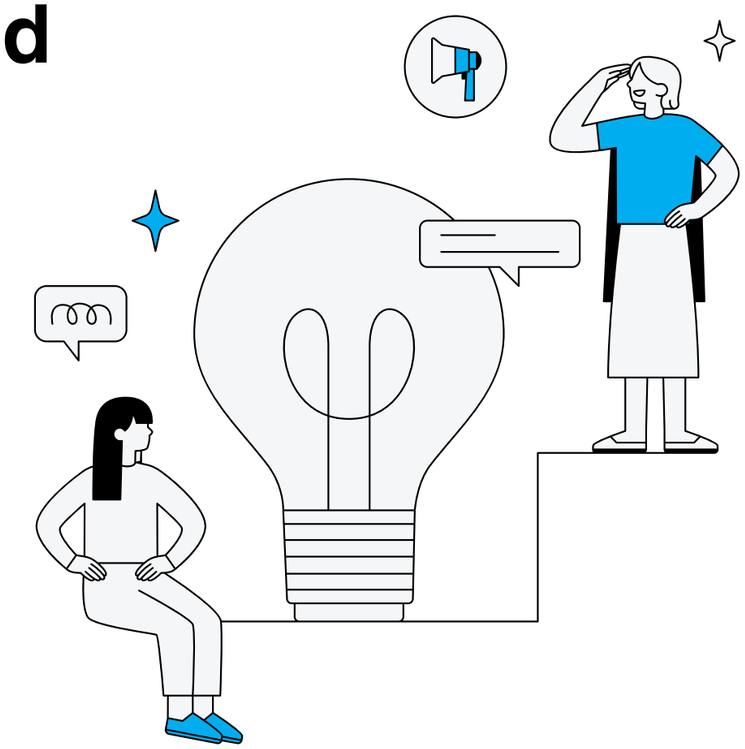
Large Language Models, like all ML models, offer no business value until they are deployed and their outputs are accessible to end-users or integrated into downstream services.

Once your LLM is fine-tuned, follow these steps to deploy it:

- Save the Model: Serialize your trained LLM using formats like TensorFlow SavedModel or PyTorch TorchScript.
- Develop an API: Create a service (such as a RESTful API) for users or applications to interact with your LLM.
- Scale Appropriately: Adjust your deployment to handle increased load by scaling horizontally or vertically.
- Monitor and Maintain: Continuously check your LLM's performance and update or retrain as necessary.

Reference : https://www.linkedin.com/pulse/deploying-large-language-models-llm-comprehensive-johnson-mba-2uycc/

Key Considerations for Framework and Backend Choices in NLP Model Deployment



1.Framework Choice Impacts Deployment: Selecting the right NLP framework affects model deployment, with Python-based tools like Scikit-learn, SpaCy, and PyTorch being popular for their robust integration and extensive ecosystem.

2.Backend Frameworks for Integration: Using backend frameworks in the same language as the model simplifies deployment. Flask is great for quick prototyping, while Django is better for complex, API-driven production systems.

3.Specialized Deployment Solutions: Tools like HuggingFace and Novetta’s AdaptNLP provide streamlined deployment options, offering scalable compute backends and Docker-based REST API microservices for efficient model deployment.

Reference: https://neptune.ai/blog/deploy-nlp-models-in-production:

Navigating the Complexities of Deploying and Managing LLMs in Production

Resource requirements: LLMs demand considerable GPU, RAM, and CPU resources, complicating cost-effective management.

Performance fluctuations: LLMs' performance can vary with data changes, necessitating continuous monitoring for consistency.

Versioning complexity: Handling updates and multiple model versions requires meticulous version control and deployment.

Infrastructure setup: Configuring deployment infrastructure, especially for multiple models, is time-intensive and intricate.

Serving LLMs in API

- APIs serve as the universal language for software communication, enabling seamless data exchange between various systems and applications.

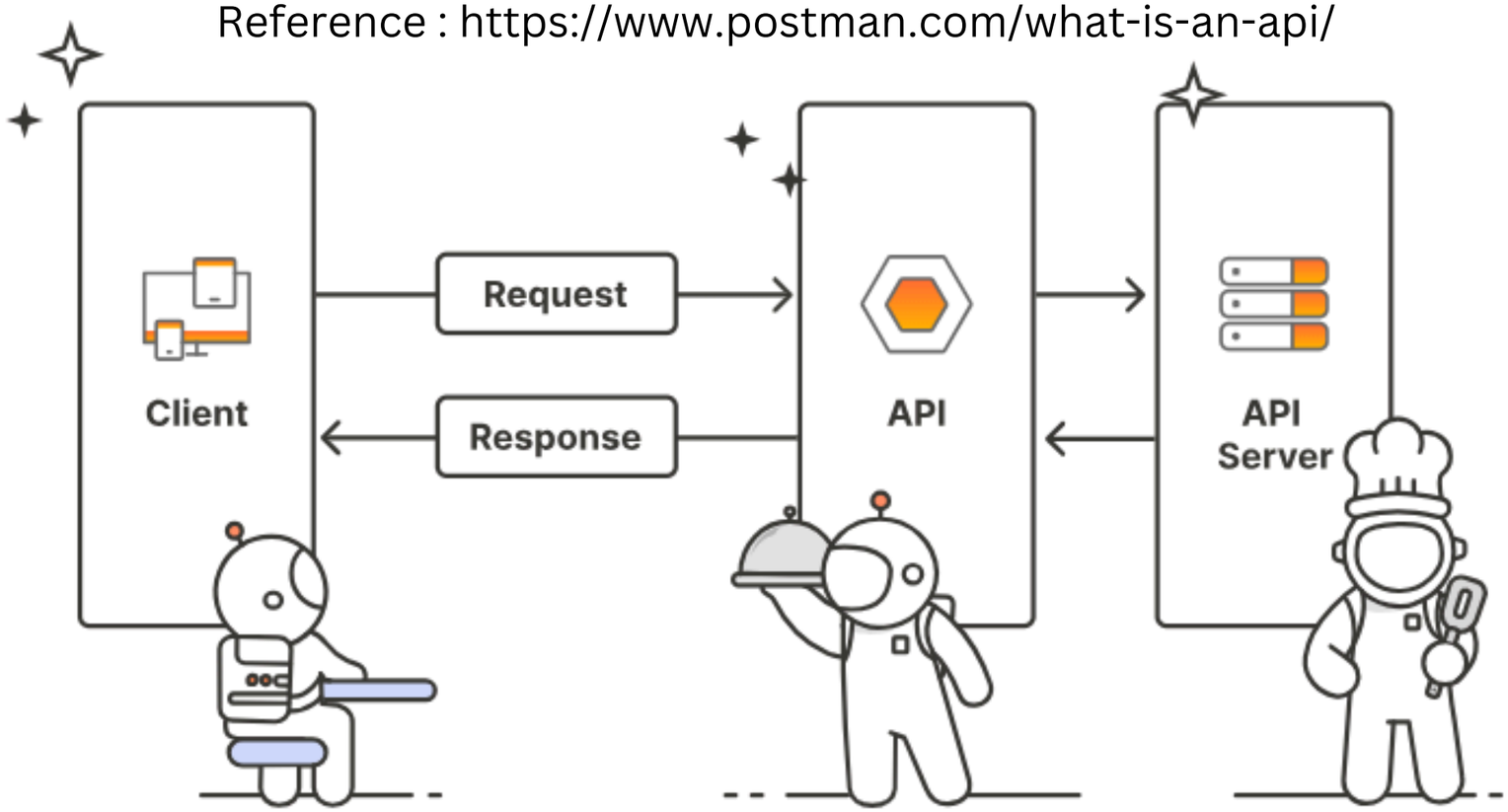
- They provide a standardized framework of rules and protocols, allowing different software entities to interact without needing to understand each other's internal workings.

- APIs are integral to modern digital ecosystems, powering web services, mobile apps, and cloud computing by facilitating inter-application communication and third-party service integration.

- Developers leverage APIs as pre-built components to accelerate software development, enhancing functionality and user experience without starting from scratch.

How APIs work?

Reference : https://www.postman.com/what-is-an-api/



- Send a Request: Your application sends a structured message (request) to the API, specifying what data or action it needs.
- API Processes the Request: The API server receives the request, authenticates it, validates the input, and performs the necessary operations (like retrieving or manipulating data).
- Receive a Response: The API sends back a structured message (response) to your application, containing the requested data or indicating the outcome of the action.
- Each step involves specific protocols and formats, such as HTTP for communication and JSON or XML for data interchange.

Steps for Developing a Customized App

1. LLM development: inference pipeline using Huggingface

Installing Pre-requisites

bash Copy code

```
pip install -U "transformers==4.40.0"
pip install accelerate bitsandbytes
```

Importing Necessary Libraries

python Copy code

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer
```

Steps for Developing a Customized App

1. LLM development: inference pipeline using Huggingface

Instantiate the Model

```
model_id = "unsloth/llama-3-8b-Instruct-bnb-4bit"


# Initialize the tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_id)

# Initialize the text generation pipeline
text_generation_pipeline = pipeline(
    "text-generation",
    model=model_id,
    tokenizer=tokenizer,
    model_kwargs={
        "torch_dtype": torch.float16,
        "quantization_config": {"load_in_4bit": True},
        "low_cpu_mem_usage": True,
    },
)
```

Steps for Developing a Customized App

1. LLM development: inference pipeline using Huggingface

Executing Model Inference Pipeline

```
model_id = "unsloth/llama-3-8b-Instruct-bnb-4bit"


# Initialize the tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_id)

# Initialize the text generation pipeline
text_generation_pipeline = pipeline(
    "text-generation",
    model=model_id,
    tokenizer=tokenizer,
    model_kwargs={
        "torch_dtype": torch.float16,
        "quantization_config": {"load_in_4bit": True},
        "low_cpu_mem_usage": True,
    },
)
```

Steps for Developing a Customized App

1. LLM development: inference pipeline using Huggingface

Executing Model Inference Pipeline

```python
# Function to generate text based on a predefined conversation template
def generate_text_from_template():
    messages = [
        {"role": "user", "content": "Can you recommend a good book for learning Python programming?"},
        {"role": "system", "content": "Sure! How about 'Python Crash Course' by Eric Matthes? It's beginner-friendly and comprehensive."},
        {"role": "user", "content": "Great! Where can I find it?"},
        {"role": "system", "content": "You can find it in most bookstores or online retailers like Amazon or Barnes & Noble."},
    ]
    prompt = pipeline.tokenizer.apply_chat_template(
        messages,
        tokenize=False,
        add_generation_prompt=True
    )
    outputs = pipeline(
        prompt,
        max_new_tokens=256,
        do_sample=True,
        temperature=0.6,
        top_p=0.9,
    )
    return outputs[0]["generated_text"][len(prompt):]
```

Steps for Developing a Customized App

2. Back-end development: Harnessing Tornado for Robust Web Services

Tornado Web Service

- Utilizing Tornado, the web framework orchestrates a web service API by defining request handlers.

- MainHandler is responsible for rendering the web page upon access, while ChatHandler facilitates communication between the user's input and the Language Model's response.

- Upon execution, the program launches a web service to consistently listen for incoming requests.

Steps for Developing a Customized App

2. Back-end development: web service using Tornado

Tornado Web Service

```python
import tornado.ioloop
import tornado.web
import json

# Store conversation histories for context management
conversation_histories = {}

class MainHandler(tornado.web.RequestHandler):

    def get(self):
        self.render("web/index.html")


class ChatHandler(tornado.web.RequestHandler):

    def post(self):
        data = json.loads(self.request.body)
        user_id = data.get('user_id', '')
        user_input = data.get('message', '')
        response = generate_response(user_id, user_input, conversation_histories, tokenizer, model)
        self.write({'response': response})


def make_app():
    return tornado.web.Application([
        (r"/", MainHandler),
        (r"/chat", ChatHandler),
        (r"/(style\.css)", tornado.web.StaticFileHandler, {"path": "web/"}),
        (r"/(script\.js)", tornado.web.StaticFileHandler, {"path": "web/"})
    ])


if __name__ == "__main__":
    app = make_app()
    app.listen(8888)
    tornado.ioloop.IOLoop.current().start()
```

Steps for Developing a Customized App

3.Front-end development: chat UI using native JS/HTML/CSS

```
import os

if not os.path.exists('web'):
    os.makedirs('web')

# Create index.html
with open("web/index.html", "w") as file:
    file.write('''
    <!DOCTYPE html>
    <html>
    <head>
        <link rel="stylesheet" type="text/css" href="style.css">
    </head>
    <body>
        <div id="chat-container">
            <div id="chat-box"></div>
            <input type="text" id="user-input" placeholder="Type a message...">
            <button onclick="sendMessage()">Send</button>
        </div>
        <script src="script.js"></script>
    </body>
    </html>
    ''')
```

Steps for Developing a Customized App

3.Front-end development: chat UI using native JS/HTML/CSS

```
# Create style.css
with open("web/style.css", "w") as file:
    file.write('''
    #chat-container {
        width: 80%;
        margin: auto;
        margin-top: 50px;
        border: 1px solid #ccc;
        padding: 10px;
    }
    #chat-box {
        height: 400px;
        overflow-y: scroll;
        border: 1px solid #ccc;
        padding: 10px;
    }
    #user-input {
        width: 80%;
    }
    ''')
```

Steps for Developing a Customized App

3.Front-end development: chat UI using native JS/HTML/CSS

```
# Create script.js
with open("web/script.js", "w") as file:
    file.write('''
    async function sendMessage() {
        const userInput = document.getElementById("user-input").value;
        const chatBox = document.getElementById("chat-box");

        const userMessageDiv = document.createElement("div");
        userMessageDiv.textContent = "User: " + userInput;
        chatBox.appendChild(userMessageDiv);

        const response = await fetch('/chat', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify({ user_id: 'user1', message: userInput })
        });

        const data = await response.json();
        const botMessageDiv = document.createElement("div");
        botMessageDiv.textContent = "Bot: " + data.response;
        chatBox.appendChild(botMessageDiv);

        document.getElementById("user-input").value = '';
    }
    ''')
```

Steps for Developing a Customized App

4. Deployment: proxy server using Ngrok

```
!ngrok config add-authtoken '2glcxxxxxxxxxxxrxxxxxxxxxx_xxxxxxxxxxxxxxxxxxxx'
```

```
!ngrok http http://localhost:8888
```

Reference : https://towardsdatascience.com/four-simple-steps-to-build-a-custom-self-hosted-llama3-application-8d1ef139b36f,https://nlpcloud.com/how-to-install-and-deploy-llama-3-into-production.html