INTRODUCTION TO LLMS

SESSION-01

MODULE-01

PRESENTER

AI SPECIALIST

DR DIEGO CORONA-LOPEZ

CLDiego/SPE_GeoHackathon_2025.git

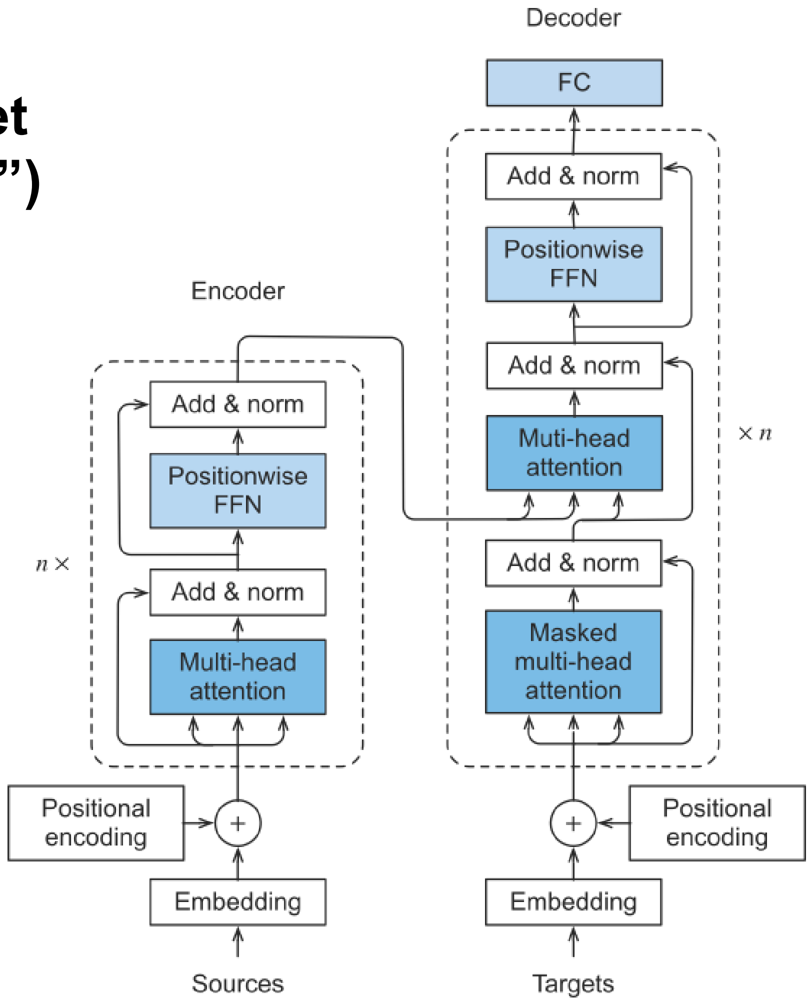
TRANSFORMERS

Neural network architecture introduced in 2017 (Vaswani et al., “Attention is All You Need”)

Replaced recurrence with attention mechanisms, making training faster and enabling scaling.

Key Features:

- Processes sequences in parallel (not step-by-step like RNNs).
- Learns relationships between tokens via self-attention.
- Scales efficiently → foundation for today’s LLMs (GPT, BERT, etc.).



Original architecture used for translation. Modern LLMs often drop the decoder or encoder depending on the task

**(BERT = encoder-only,
GPT = decoder-only)**

Enabled breakthroughs in NLP, scientific text processing, and coding assistants.

Core of models you’ll use today (**LLaMA, Phi, GPT2**).

Applications in Geoscience:

- Parsing long technical reports.
- Q&A over structured + unstructured geological data.
- Pattern recognition in sequences (seismic logs, time series).

TOKENIZERS

A token is usually a subword or character chunk

Tokenization reduces vocabulary size and helps model handle rare/unknown words

LLMs don't see raw text → they see **tokens** (sub-word units)

Even characters or byte-pair **encodings**

Different models use **different vocabularies** (LLaMA vs GPT vs Gemma).

"Seismic data is complex"

↓

["Se", "ismic", " data", " is", " complex"]

↓

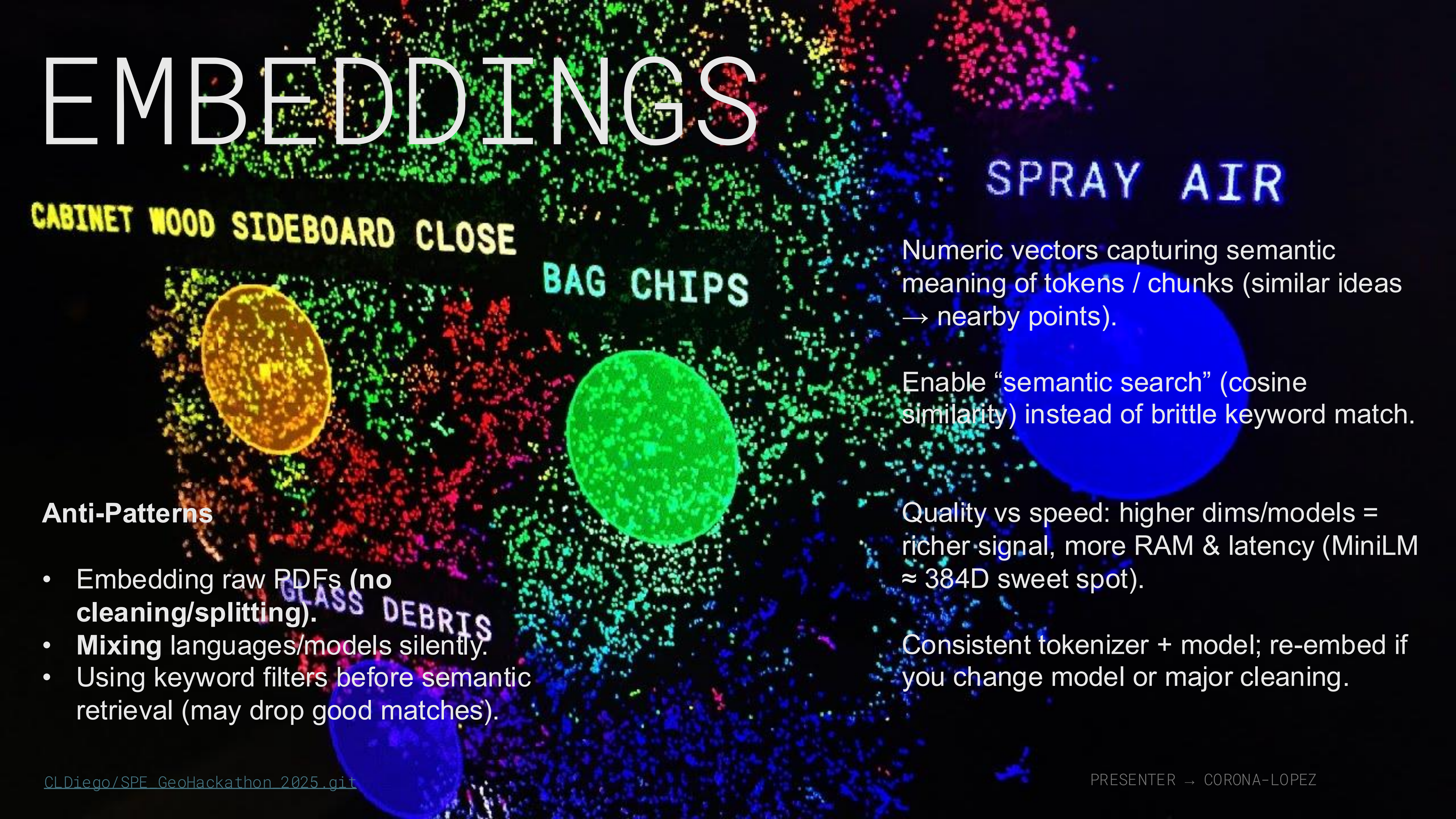
[1421, 560, 782, 91, 204]

```python
from transformers import AutoModel, AutoTokenizer
import torch

# Load a small model for embeddings
model_name = "sentence-transformers/all-MiniLM-L6-v2"
tokenizer = AutoTokenizer.from_pretrained(model_name)

sample_text = "Seismic data is complex"
tokens = tokenizer.tokenize(sample_text)
token_ids = tokenizer.convert_tokens_to_ids(tokens)
```

More compact tokenization = fewer tokens → faster inference

CLDiego/SPE_GeoHackathon_2025.git

EMBEDDINGS



Numeric vectors capturing semantic meaning of tokens / chunks (similar ideas → nearby points).

Enable “semantic search” (cosine similarity) instead of brittle keyword match.

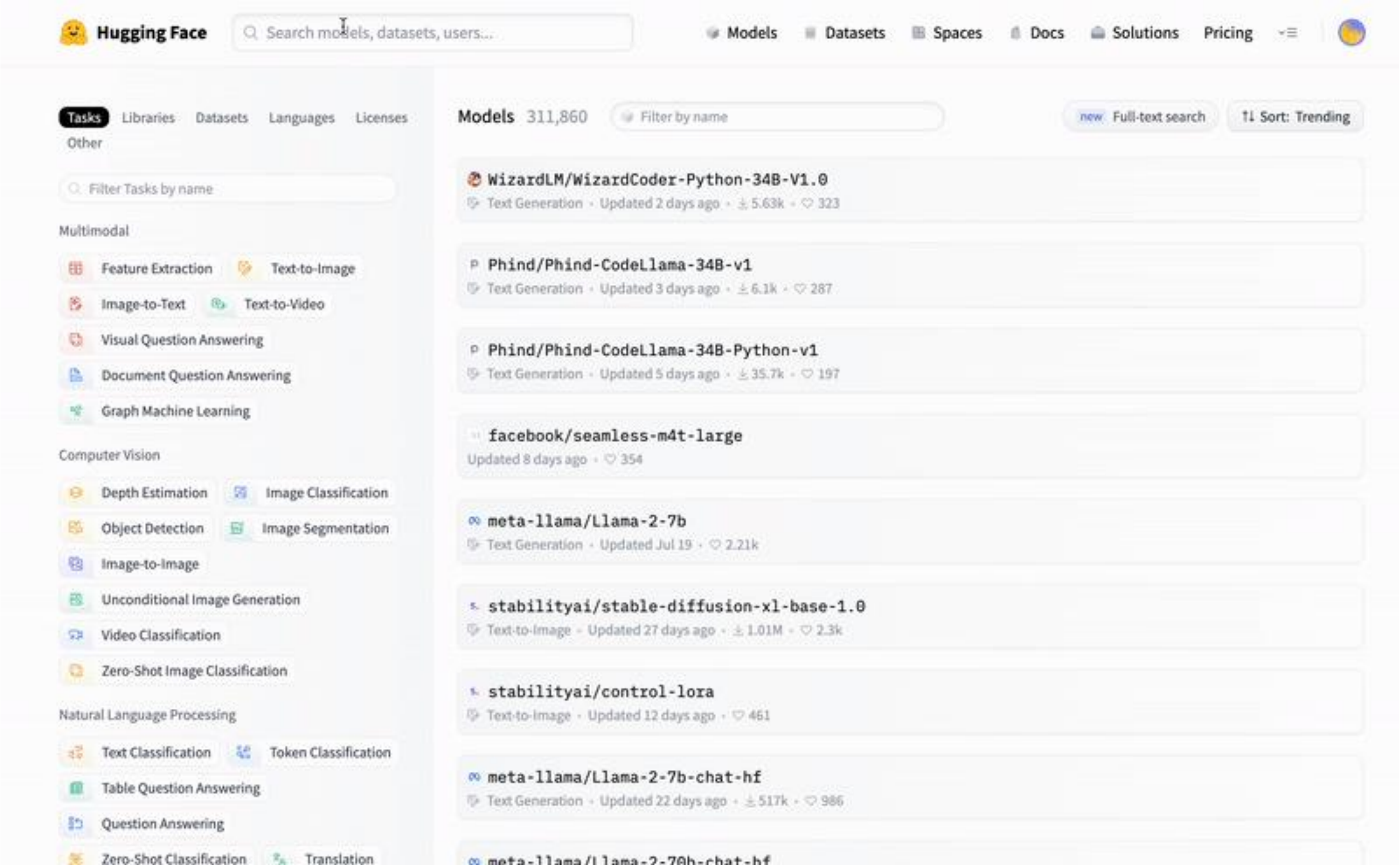
Quality vs speed: higher dims/models = richer signal, more RAM & latency (MiniLM ≈ 384D sweet spot).

Consistent tokenizer + model; re-embed if you change model or major cleaning.

Anti-Patterns

- Embedding raw PDFs **(no cleaning/splitting).**
- **Mixing** languages/models silently.
- Using keyword filters before semantic retrieval (may drop good matches).

HUGGING-FACE



Platform services: Spaces (deploy demos), Inference Endpoints / TGI, AutoTrain, Hub versioning + PRs, Collections, Community discussions.

Model Hub: thousands of open & gated models (LLMs, embeddings, vision, audio); read model cards (license, intended use, limits).

Core Python libs:

- transformers (inference/fine-tune)
- datasets (streaming data)
- tokenizers (fast Rust)
- peft (LoRA/adapters)
- accelerate (multi-device)
- diffusers (generative media)
- evaluate & metrics

LOADING HF MODELS

```python
# Auth (only if gated / private)
from huggingface_hub import login
# login(token=HF_TOKEN)  # or set env HF_TOKEN

from transformers import AutoTokenizer, AutoModelForCausalLM
import torch

MODEL_ID = "distilgpt2"          # small demo model
REVISION = None                  # e.g. "main" or a commit hash
device = (
    "cuda" if torch.cuda.is_available()
    else "mps" if getattr(torch.backends, "mps", None) and torch.backends.mps.is_available()
    else "cpu"
)

tokenizer = AutoTokenizer.from_pretrained(MODEL_ID, revision=REVISION)
model = AutoModelForCausalLM.from_pretrained(
    MODEL_ID,
    revision=REVISION,
    torch_dtype=torch.float16 if device in ("cuda","mps") else torch.float32,
    device_map="auto" if device == "cuda" else None,
)

# Safety: define pad token if missing
if tokenizer.pad_token_id is None:
    tokenizer.pad_token = tokenizer.eos_token
```

Purpose: Get tokenizer + model locally (reproducible, safe).

Steps:

- (Optional) Authenticate (only for gated/private) with HF_TOKEN.
- Pick a model id + (optional) pinned revision (commit hash or tag).
- Load tokenizer first, then model (set device + dtype).
- Ensure pad/eos tokens for generation.

Tips:

- Pin revision for reproducibility.
- Smaller model → faster demos; larger model → better knowledge.
- Use device_map="auto" (GPU) or 4-bit (bitsandbytes) for memory saving

TEXT-GENERATION

.py

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

MODEL_ID = "distilgpt2"
tokenizer = AutoTokenizer.from_pretrained(MODEL_ID)
model = AutoModelForCausalLM.from_pretrained(MODEL_ID)
if tokenizer.pad_token_id is None:
    tokenizer.pad_token = tokenizer.eos_token

def generate(prompt: str):
    inputs = tokenizer(prompt, return_tensors="pt")
    with torch.no_grad():
        out_ids = model.generate(
            **inputs,
            max_new_tokens=60,
            temperature=0.7,
            top_p=0.9,          # or top_k=50
            do_sample=True,
            repetition_penalty=1.1,
            no_repeat_ngram_size=2,
            eos_token_id=tokenizer.eos_token_id,
            pad_token_id=tokenizer.pad_token_id,
        )
    return tokenizer.decode(out_ids[0], skip_special_tokens=True)

print(generate("Reservoir characterization involves"))
```

Flow: prompt → tokenize → generate → decode.

Core parameters (generation quality vs speed):

- max_new_tokens: cap length (latency control).
- temperature: randomness (0.1–0.3 factual, 0.7+ creative).
- top_p (nucleus): keep minimal probability mass (e.g. 0.9).
- top_k: limit candidate tokens (use one of top_p or top_k).
- do_sample=True: enable stochastic decoding (else greedy).
- repetition_penalty / no_repeat_ngram_size: reduce loops.
- eos_token_id, pad_token_id: clean termination.

Tuning heuristics:

- Too random → lower temperature or top_p.
- Repetition → increase penalty or n-gram constraint.
- Truncation → raise max_new_tokens (watch context window).
- Deterministic baseline → set do_sample=False (greedy).

LANG-CHAIN

PRESENTER

SESSION-01

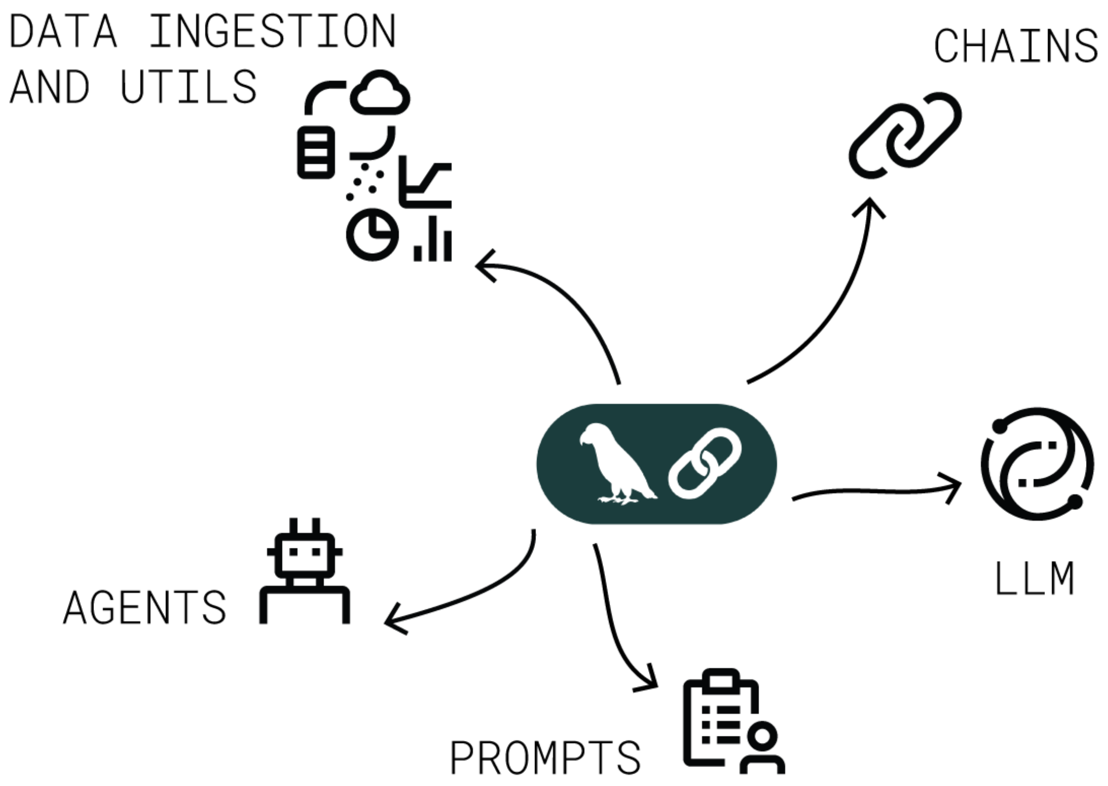
MODULE-02

AI SPECIALIST

DR DIEGO
CORONA-LOPEZ

CLDiego/SPE_GeoHackathon_2025.git

101 BUILDING BLOCKS



An open-source framework providing modular, reusable components.

Core Philosophy: Composability.

We'll use the LangChain Expression Language (LCEL) and the pipe (|) operator to connect them.

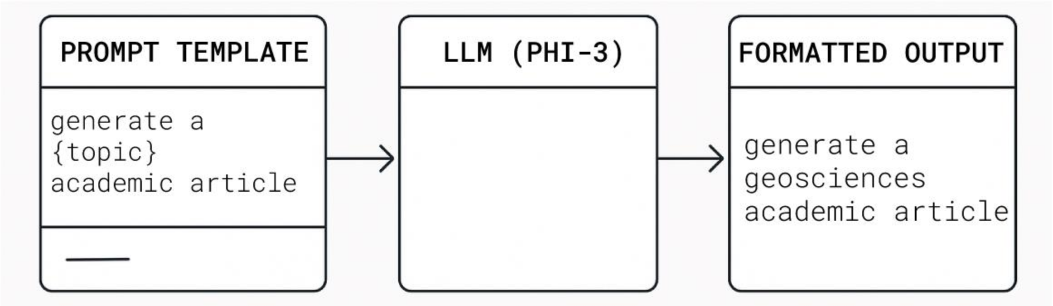
Key Components:

- LLM Wrappers: Standard interface for any model (Hugging Face, OpenAI, etc.).
- Prompt Templates: Create dynamic, reusable prompts.
- Chains: The sequence that ties everything together.
- Memory: To remember conversation history.

STANDARDIZATION

Prompt Templates: Structure your prompts with placeholders for dynamic content. This is crucial for reliability

LLM Wrappers: Swap between different LLMs (e.g., Phi-3 to Llama-3) with minimal code changes.



.py

```python
from langchain_huggingface import ChatHuggingFace
from langchain_core.prompts import ChatPromptTemplate

# 1. The LLM Wrapper
chat_model = ChatHuggingFace(llm=...) # Hides complex setup

# 2. The Prompt Template
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are Dr. GeoBot, a geoscience expert."),
    ("human", "{question}")
])
```

LCEL CHAINS

.py

```python
from langchain_core.output_parsers import StrOutputParser

# The "chain" is created using the pipe operator
chain = prompt | chat_model | StrOutputParser()

# Invoking the chain runs the whole sequence
response = chain.invoke({"question": "What is porosity?"})
print(response)
```

Chains are the core of LangChain, defining the sequence of operations

LangChain Expression Language (LCEL) uses the pipe (|) operator to make this intuitive and powerful

It reads like a data pipeline:
input -> prompt -> model -> output.

CONVERSATIONAL CHAINS

By default, chains are stateless.

Memory allows a chain to remember previous interactions.

We'll use RunnableWithMessageHistory to automatically manage the conversation.

- It works by adding a MessagesPlaceholder to your prompt and tracking history based on a session_id

.py

```python
from langchain.prompts import MessagesPlaceholder
from langchain_core.runnables.history import RunnableWithMessageHistory

# 1. Add a placeholder for history
conv_prompt = ChatPromptTemplate.from_messages([
    ("system", "You are Dr. GeoBot..."),
    MessagesPlaceholder(variable_name="history"), # History goes here
    ("human", "{question}")
])

# 2. Wrap the chain to manage history
chain_with_memory = RunnableWithMessageHistory(
    conv_prompt | chat_model | StrOutputParser(),
    get_session_history, # Function to get/create a history object
    ...
)
```

AGENT UI

PRESENTER

SESSION-01

MODULE-03

AI SPECIALIST

DR DIEGO
CORONA-LOPEZ

CLDiego/SPE_GeoHackathon_2025.git

CHAT-AGENT

To build a clean UI, we first need to organize our code.

```python
class ModernGeoscienceChatAgent:
    def __init__(self, chat_model):
        # ... setup chain, prompt, store ...

    def chat(self, question: str, session_id: str) → str:
        # ... invoke the conversational chain ...
        return response

    def clear_memory(self, session_id: str):
        # ... clear history for the session ...
```

We'll create a ModernGeoscienceChatAgent class to hold our:

- ChatModel
- PromptTemplate
- The conversational chain
- Session history store and management methods (chat, clear_memory).

This makes our logic portable and easy to plug into any interface. It's a key step towards building more complex systems.

GRADIO UI

The gr.ChatInterface is the easiest way to build a chatbot UI.

It requires a single function that takes a message and history as input and returns the bot's response.

We'll create a respond function that calls our gradio_agent.chat() method.

.py

```python
import gradio as gr

# This function connects our agent to the UI
def respond(message, history):
    bot_response = gradio_agent.chat(message, current_session)
    history.append((message, bot_response))
    return "", history # Return empty string to clear input box

# Create and launch the UI with one line!
ui = gr.ChatInterface(
    fn=respond,
    title="Dr. GeoBot"
)
ui.launch()
```

WHAT'S NEXT?

Our bot can talk, but it can't *do* things like perform calculations or search for live data.

- **Tools** are functions an LLM can call to interact with the world (e.g., a calculator, a database).
- An **Agent** is a system that uses an LLM's reasoning to decide *which* tool to use to answer a question.
- This is a powerful concept we will explore in-depth in **Session 2**.