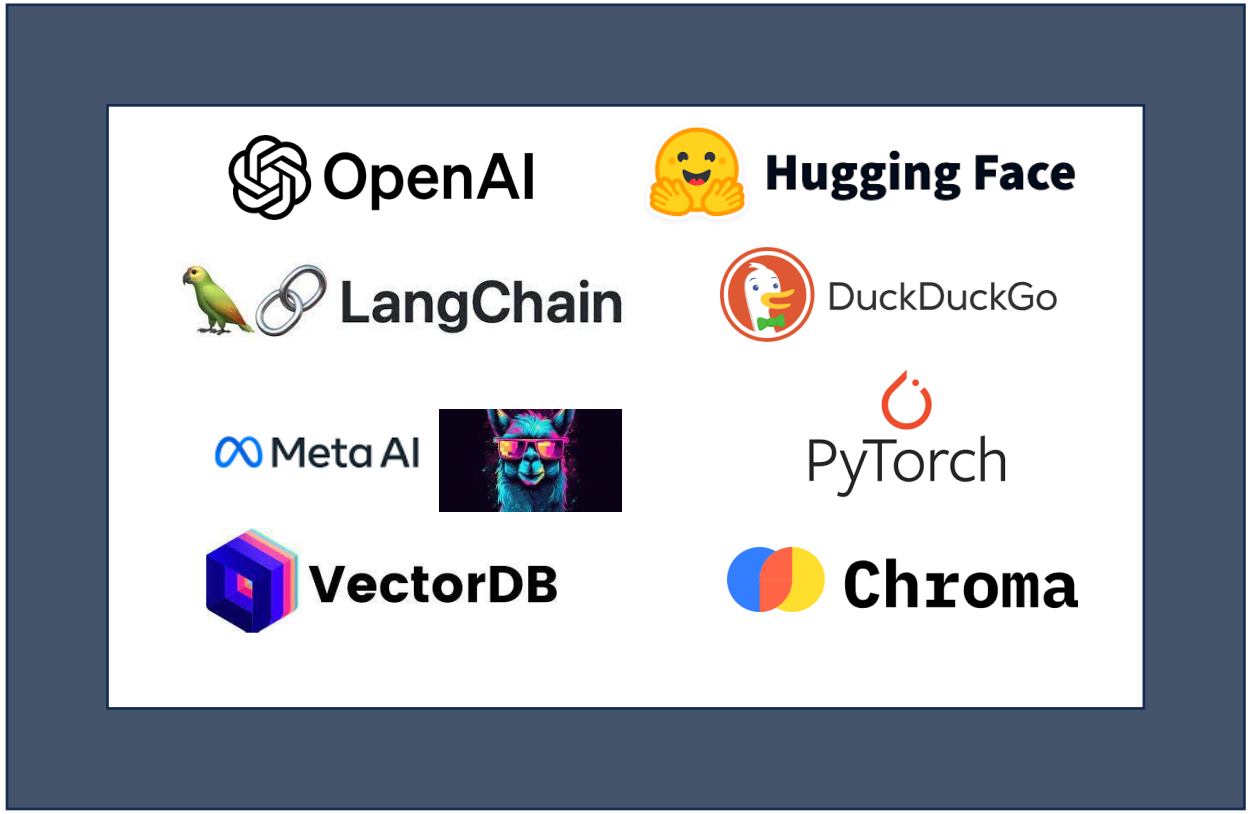
LLM-Zero-to-Hundred

- LLM tutorials (Text embedding, Function calling)
- Chatbots:
  - RAG-GPT
  - WebGPT
  - DeepWebQuery
  - Multimodal chatbot
- LLM fine-tuning:
  - Full fine tuning
  - PEFT (Parameter efficient fine tuning)
- Pretraining LLMs

gradio
Streamlit
Chainlit
Flask



Text Embedding
&
Basic RAG

Content

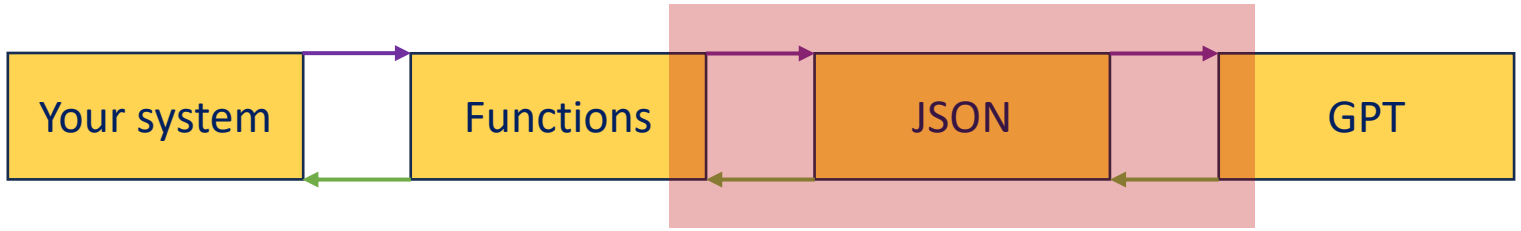
1. Text embedding
2. Vector search: Cosine similarity
3. Simple RAG:
    Main steps:
        - Create VectorDB
        - Perform vector search on the query and the vectordb
        - Retrieve the most relevant content
        - Use a LLM to check the query and the result and provide the answer.

Two notebooks:

- OpenAI: **text-embedding-ada-002** with **GPT 3.5 Turbo**
- Open source: **BAAI/bge-large-zh-v1.5** with **LLAMA 2 7b**

Function Calling

Key concept



Retrieval Augmented Generation (RAG-GPT)

RAG: Retrieval Augmented Generation

Data Ingestion

Loading documents

Chunking

Embedding

Indexing

Content retrieval

Query

Embedding

Search index

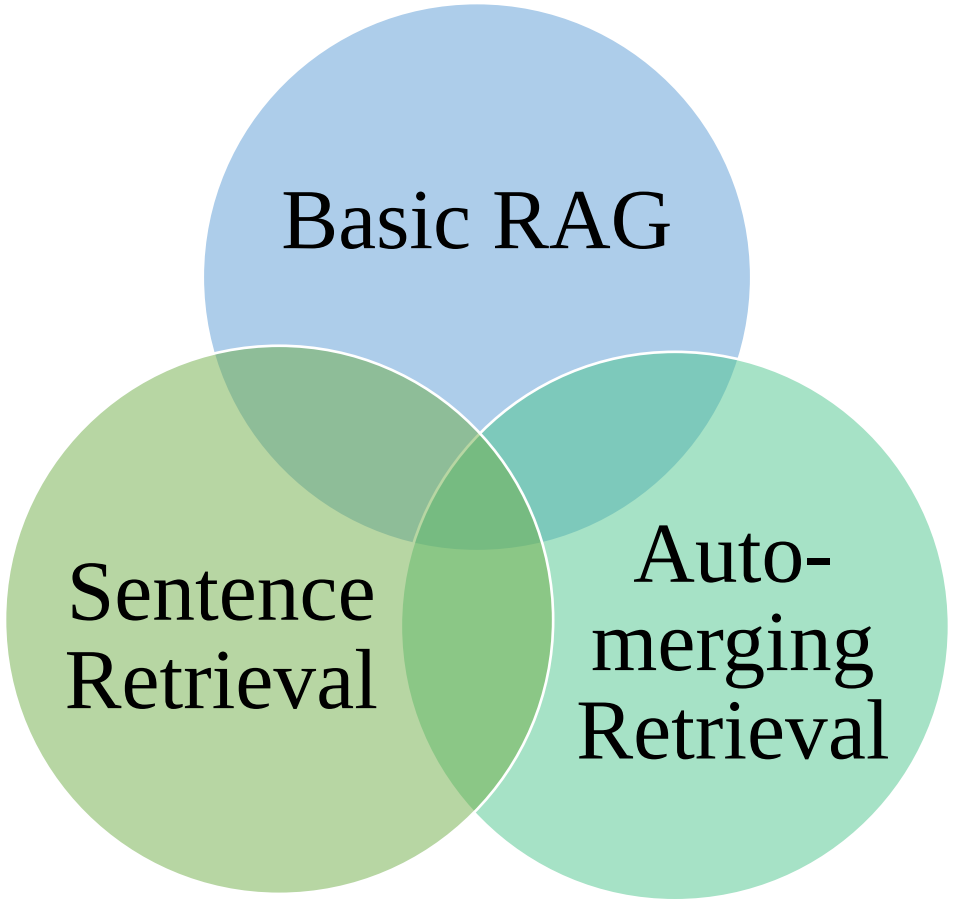
Retrieve top k chunks

Synthesis

Input: LLM instruction + query + retrieved contents + chat history

Output: LLM response

RAG Techniques



The Conventional Method

First chunk

Overlap

Second chunk

The Transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, shown in the left and right halves of Figure 1, respectively.

3.1 Encoder and Decoder Stacks

Encoder: The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection [11] around each of the two sub-layers, followed by layer normalization [1]. That is, the output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$, where $\mathrm{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Decoder: The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

3.2 Attention

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum

Sentence Retrieval

The Transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, shown in the left and right halves of Figure 1, respectively.

3.1 Encoder and Decoder Stacks

Encoder: The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection [11] around each of the two sub-layers, followed by layer normalization [1]. That is, the output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$, where $\mathrm{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Passes to LLM

Search result

Decoder: The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

3.2 Attention

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum

Auto-merging Retrievals

Parent node

A single convolutional layer with kernel width $k < n$ does not connect all pairs of input and output positions. Doing so requires a stack of $O(n/k)$ convolutional layers in the case of contiguous kernels, or $O(log_k(n))$ in the case of dilated convolutions [18], increasing the length of the longest paths between any two positions in the network. Convolutional layers are generally more expensive than recurrent layers, by a factor of k. Separable convolutions [6], however, decrease the complexity considerably, to $O(k \cdot n \cdot d + n \cdot d^2)$. Even with $k = n$, however, the complexity of a separable convolution is equal to the combination of a self-attention layer and a point-wise feed-forward layer, the approach we take in our model.

Child node 1

As side benefit, self-attention could yield more interpretable models. We inspect attention distributions from our models and present and discuss examples in the appendix. Not only do individual attention heads clearly learn to perform different tasks, many appear to exhibit behavior related to the syntactic and semantic structure of the sentences.

Child node 2

5 Training

This section describes the training regime for our models.

5.1 Training Data and Batching

We trained on the standard WMT 2014 English-German dataset consisting of about 4.5 million sentence pairs. Sentences were encoded using byte-pair encoding [3], which has a shared source-target vocabulary of about 37000 tokens. For English-French, we used the significantly larger WMT 2014 English-French dataset consisting of 36M sentences and split tokens into a 32000 word-piece vocabulary [38]. Sentence pairs were batched together by approximate sequence length. Each training batch contained a set of sentence pairs containing approximately 25000 source tokens and 25000 target tokens.

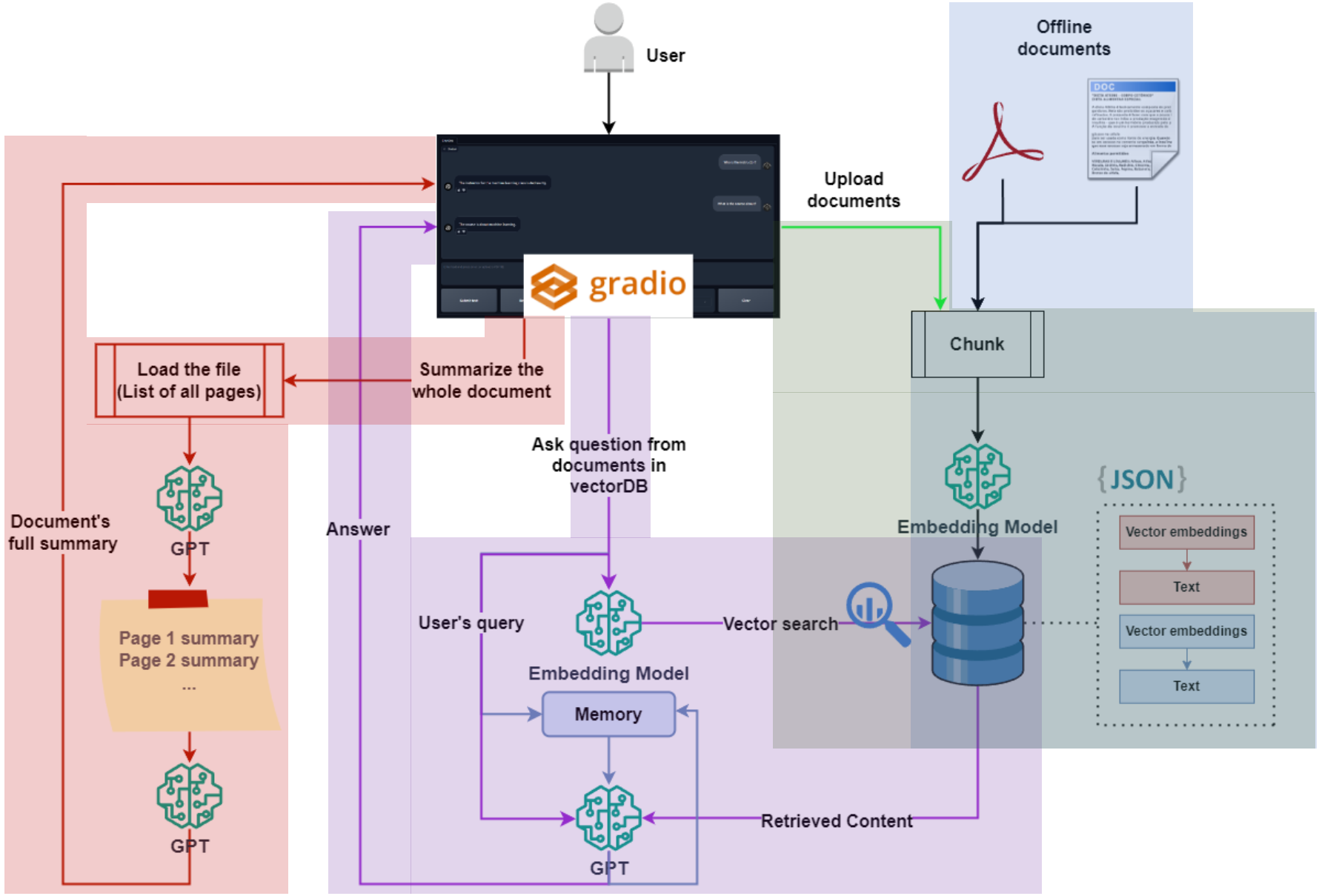
Child node 3

5.2 Hardware and Schedule

We trained our models on one machine with 8 NVIDIA P100 GPUs. For our base models using the hyperparameters described throughout the paper, each training step took about 0.4 seconds. We trained the base models for a total of 100,000 steps or 12 hours. For our big models,(described on the bottom line of table 3), step time was 1.0 seconds. The big models were trained for 300,000 steps (3.5 days).

Child node 4

Project Schema



Optimization strategies

1. Model selection: LLM - Embedding
2. LLM instruction
3. LLM configs: temperature, token limit, etc.
4. Context length (LLM Instruction, Query, Retrieved content, history)
5. Selection/designing the RAG technique
6. Chunk size, and overlap – Child and parent size – sentence window size

Deployment considerations

Memory - privacy

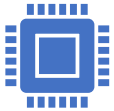
Databases and data flow management (adding, updating, and removing documents)

Testing

Security (Data leakage, Authentication, etc.)

Special scenarios (e.g: A document that needs to be included in all searches)

Deployment – pipeline



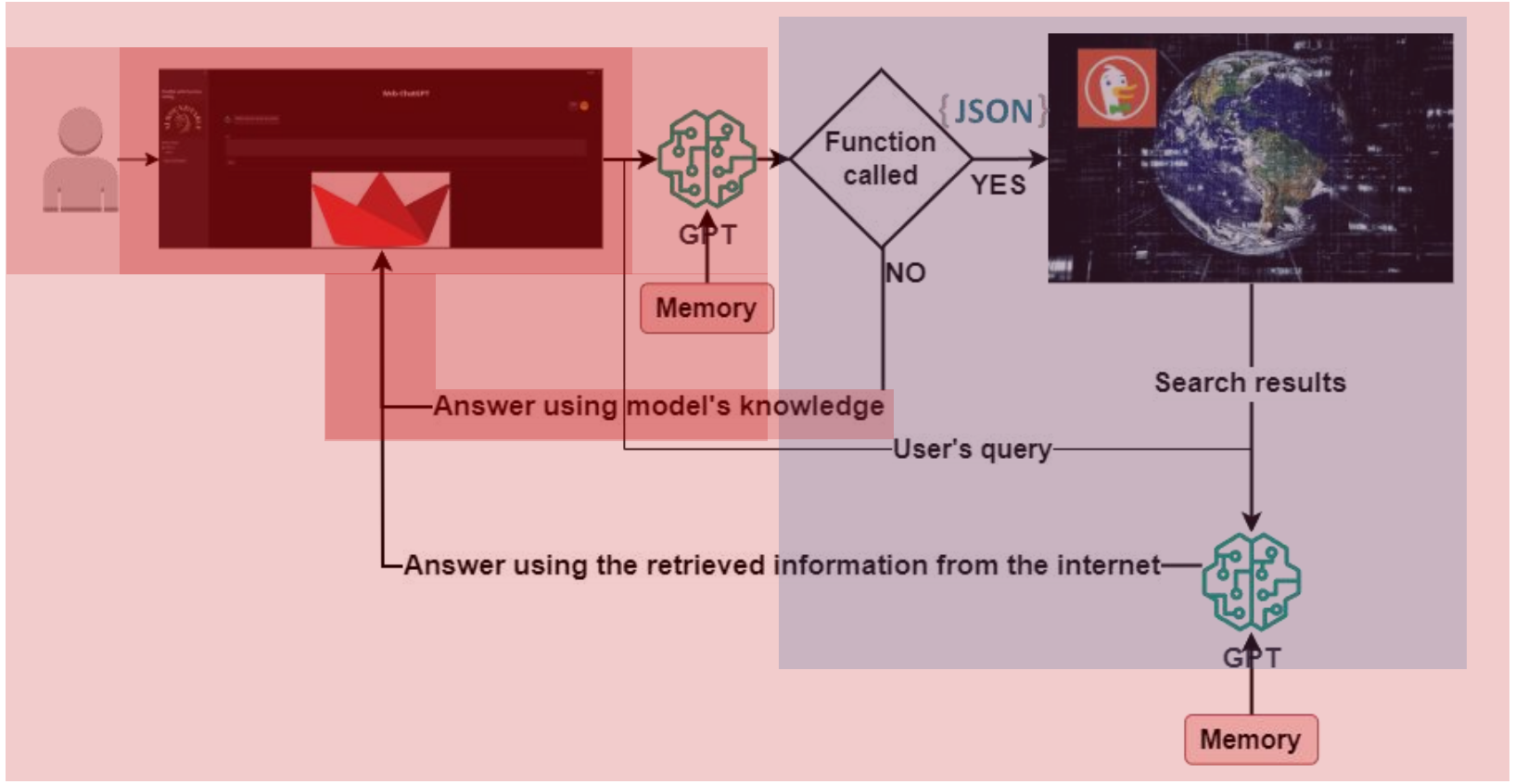
Latency, Scale, Computation, Server, Price

Monitor and Log

Feedback

WebGPT:
A GPT Agent Connected to the Internet

Addressing the LLM knowledge cut-off with real-time web search using GPT agents and function calling



Further steps

1. Agent’s complexity
2. Better system design (e.g: memory)
3. Adding more features (e.g: RAG with a specific website)

Deployment considerations

Memory - privacy

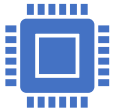
Databases and data flow management (adding, updating, and removing documents)

Testing

Security (Data leakage, Authentication, legal concerns, etc.)

Special scenarios (e.g: illegal websites)

Deployment – pipeline



Latency, Scale, Computation, Server, Price

Monitor and Log

Feedback

DeepWebQuery (Combining RAG-GPT and WebGPT)

Project Schema